ORIENTATION DE L'EFFORT DES TESTS UNITAIRES DANS LES SYSTÈMES ORIENTÉS OBJET: UNE APPROCHE BASÉE SUR LES MÉTRIQUES LOGICIELLES

Thèse

Fadel TOURE

Doctorat en informatique
Philosophiæ doctor (Ph.D.)

Québec, Canada

ORIENTATION DE L'EFFORT DES TESTS UNITAIRES DANS LES SYSTÈMES ORIENTÉS OBJET: UNE APPROCHE BASÉE SUR LES MÉTRIQUES LOGICIELLES

Thèse

Fadel TOURE

Sous la direction de :

Luc LAMONTAGNE, directeur de recherche
Mourad BADRI, codirecteur de recherche

RÉSUMÉ

Les logiciels actuels sont de grandes tailles, complexes et critiques. Le besoin de qualité exige beaucoup de tests, ce qui consomme de grandes quantités de ressources durant le développement et la maintenance de ces systèmes. Différentes techniques permettent de réduire les coûts liés aux activités de test. Notre travail s'inscrit dans ce cadre, est a pour objectif d'orienter l'effort de test vers les composants logiciels les plus à risque à l'aide de certains attributs du code source. À travers plusieurs démarches empiriques menées sur de grands logiciels open source, développés avec la technologie orientée objet, nous avons identifié et étudié les métriques qui caractérisent l'effort de test unitaire sous certains angles. Nous avons aussi étudié les liens entre cet effort de test et les métriques des classes logicielles en incluant les indicateurs de qualité. Les indicateurs de qualité sont une métrique synthétique, que nous avons introduite dans nos travaux antérieurs, qui capture le flux de contrôle ainsi que différentes caractéristiques du logiciel. Nous avons exploré plusieurs techniques permettant d'orienter l'effort de test vers des composants à risque à partir de ces attributs de code source, en utilisant des algorithmes d'apprentissage automatique. En regroupant les métriques logicielles en familles, nous avons proposé une approche basée sur l'analyse du risque des classes logicielles. Les résultats que nous avons obtenus montrent les liens entre l'effort de test unitaire et les attributs de code source incluant les indicateurs de qualité, et suggèrent la possibilité d'orienter l'effort de test à l'aide des métriques.

ABSTRACT

Current software systems are large, complex and critical. The need for quality requires a lot of tests that consume a large amount of resources during the development and the maintenance of systems. Different techniques are used to reduce the costs of testing activities. Our work is in this context. It aims to guide the unit testing effort distribution on the riskiest software components using the source code attributes. We conducted several empirical analyses on different large object-oriented open source software systems. We identified and studied several metrics that characterize the unit testing effort according to different perspectives. We also studied their relationships with the software class metrics including quality indicators. The quality indicators are a synthetic metric that we introduced in our previous work. It captures control flow and different software attributes. We explored different approaches for unit testing effort orientation using source code attributes and machine learning algorithms. By grouping software metrics, we proposed an effort orientation approach based on software class risk analysis. In addition to the significant relationships between testing metrics and source code attributes, the results we obtained suggest the possibility of using source code metrics for unit testing effort orientation.

TABLE DES MATIÈRES

LISTE DES TABLES

LISTE DES FIGURES

À ma mère Fatimata DIAGANA.

18 octobre 1943 - 06 octobre 2010

REMERCIEMENTS

Je remercie mes directeurs de thèse Mourad BADRI et Luc LAMONTAGNE pour le soutien et les encouragements, le dévouement, la patience et toutes ces années de recherche et de publications. Merci d'avoir partagé votre expérience. Je vous serais, pour toujours, reconnaissant.

Un grand merci à mon épouse Mairy SY pour la compréhension, le soutien moral et tout l'amour qu'elle me porte.

Je remercie également les membres de ma famille pour leurs prières, leurs encouragements et leur soutien sans faille. Un merci particulier à ceux qui sont présents aujourd'hui à Québec: Dr Ibra TOURE, Mamadou TOURE et Racine BA.

Merci à mon père pour tous les sacrifices consentis pour nos études.

Je dédie ce mémoire à ma chère mère Fatimata DIAGANA qui disparue un jour de cours, le 06 octobre 2010. Je venais à peine de débuter ce doctorat.

Que ton âme repose en paix et que le Bon Dieu t'accueille dans son paradis éternel. Aujourd'hui toutes mes pensées vont vers toi. Merci de m'avoir encouragé dans les études et d'avoir laissé ton travail pour m'accompagner à ma difficile première journée d'école.

Merci Néné.

CHAPITRE 1. INTRODUCTION

1.1 Introduction

Les systèmes logiciels sont devenus incontournables dans nos sociétés post-industrielles. Ils interviennent dans presque tous les secteurs d'activité. Ils permettent d'automatiser des traitements répétitifs dans certains cas, mais aussi prennent de plus en plus de décisions critiques [1,2]. L'augmentation et la grande vitesse d'acquisition de quantités phénoménales de données font du logiciel un outil indispensable pour le traitement en temps réel de l'information collectée [3]. En conséquence, les systèmes logiciels ont atteint des tailles et des niveaux de complexité engendrant plusieurs problèmes qui deviennent de plus en plus difficiles à gérer. Dans le même temps, les exigences de qualité logicielles sont devenues très élevées. Il s'en est suivi une complexification du processus de développement à différents niveaux.

Pour répondre à cette nouvelle réalité, les approches de développement ont dû évoluer et s'adapter aux nouvelles réalités pour gérer les équipes, les délais et les changements. L'apparition de méthodes de développements agiles [4,5] a permis de s'adapter rapidement aux spécifications changeantes et de diviser le processus de développement en itérations plus petites et maitrisables. Dans le même sens, les environnements de développement ont aussi grandement évolué, précédés dans cela par les technologies de programmation.

Ainsi, la technologie orientée objet (OO) [6] qui avait vu le jour plusieurs années plus tôt s'est vite imposée comme standard dans le développement logiciel. L'OO constitue, en fait, une évolution majeure des langages de programmation. Le paradigme a introduit de nouveaux concepts tels que les classes permettant de regrouper les données et les fonctionnalités qui les manipulent dans des unités cohésives, l'héritage permettant la réutilisation du code et le polymorphisme supportant des comportements dynamiques multiples.

Par ailleurs, les métriques logicielles apparues plus tôt [7] ont vu leur utilisation se répandre en ingénierie logicielle. Elles permettent (entre autres) de mesurer et de

quantifier différents artéfacts logiciels dans le but d'évaluer la qualité du produit et d'aider à la maitrise du processus de développement et de maintenance [8]. Au centre du processus de création du logiciel, le facteur humain joue un rôle majeur. Sous la pression des délais, des exigences de qualité et de la complexité des architectures logicielles, les développeurs doivent écrire des centaines de milliers de lignes de code réparties dans des milliers de classes OO qui communiquent selon une architecture précise, logique, compréhensible et maintenable. Dès lors, la présence de faute devient inévitable. Alors que certaines fautes sont découvertes lors du développement entrainant (entre autres) des délais dans le processus de développement, d'autres en revanche sont découvertes suite à des défaillances durant la phase d'exploitation, impliquant des coûts financiers qui peuvent être considérables [9].

Afin de prévenir les fautes, les tests ont été inclus à part entière dans le cycle de vie du logiciel. Ceux-ci permettent de vérifier et de valider les logiciels. Il existe plusieurs types de test s'appliquant à diverses phases du développement et de la maintenance [10]. Leur objectif est de découvrir les fautes présentes dans le logiciel. Sachant que la découverte tardive des fautes entraine des coûts de maintenance plus élevés [8,11], le test est devenu de plus en plus précoce dans le cycle de développement. Dans ce contexte, les tests unitaires permettent de valider individuellement les classes logicielles des systèmes orientés objet. De nouveaux processus de développement pilotés par les tests TDD (Test Driven Development) [12] ont vu le jour et exigent des développeurs l'écriture des tests unitaires avant ou pendant les développements des classes logicielles. L'objectif des tests et de cette nouvelle approche de développement est alors de maximiser la découverte de fautes présentes dans le code, le plus tôt possible dans le cycle de développement.

Dopés par les exigences de qualité, la complexité des architectures et la taille des logiciels, les tests occupent une place de plus en plus importante dans les différentes phases du processus de développement, et font partie des étapes fortement consommatrices de ressources [13]. Il faut aussi préciser que les tests constituent une activité primordiale et incontournable dans l'assurance de la qualité des logiciels. C'est dans ce contexte que vont naître des stratégies d'optimisations des tests. En effet, tester

entièrement ces logiciels se révèle vite être irréaliste du fait de l'effort à investir, des coûts et des délais de développement et de maintenance que cela entraine. Le test est un processus exigeant et assez complexe, caractérisé par plusieurs facettes (planification, conception, conduite, etc.).

C'est ainsi que certains chercheurs vont investiguer les moyens de créer des tests plus efficaces [14], tandis que d'autres vont élaborer des techniques d'optimisation des multitudes de cas de test générés pour en améliorer le rendement [15], et d'autres encore vont explorer le développement d'approches et d'outils pour orienter les tests et cibler les composants critiques du logiciel. C'est dans ce dernier volet que s'inscrivent les investigations de cette thèse. Nous avons exploré différentes possibilités d'orientation des tests unitaires logiciels vers les composants les plus à risque en nous basant uniquement sur les métriques de code source. Cette démarche nous a conduit, entre autres, à investiguer les liens entre les métriques et l'effort de test à travers différentes analyses qui ont mené à des contributions et publications importantes.

1.2 Contexte

La taille et la complexité des logiciels actuels ont vu naître de nouveaux processus de développement adaptés. Le cycle de développement linéaire (purement séquentiel) est de plus en plus abandonné du fait principalement de sa rigidité, au profit des processus de développement itératifs à cycle plus courts, s'adaptant rapidement aux changements continus des spécifications des clients et permettant des validations continues et plus précoces. Dans ces nouveaux processus, des groupes restreints de fonctionnalités du produit final sont spécifiés, développés, testés (et intégrés) puis soumis à la validation du client. Les exigences de qualité obligent les équipes à développer des suites de tests unitaires pour ses fonctionnalités. Dans le contexte de l'orienté objet, cela se traduit par la création de plusieurs classes à chaque itération. Tandis que le TDD [12] exige l'écriture des classes tests avant le code des classes sources, les autres processus de développement à cycles courts font écrire les classes tests après le développement entier de groupes de

fonctionnalités. En conséquence, plusieurs classes sources sont écrites (regroupant quelques fonctionnalités) et doivent être testées unitairement.

D'autres logiciels continuent d'être utilisés dans des organisations alors qu'ils sont supplantés par des systèmes plus modernes. On parle de systèmes hérités. L'obsolescence de ces systèmes et leur criticité les rendent difficilement remplaçables sans engendrer des projets coûteux et risqués. Ces systèmes peuvent présenter plusieurs composants qui sont peu ou pas testés selon les normes de qualité actuelles. Pour la maintenance de ces systèmes, les équipes de développement doivent réutiliser, étendre et modifier les classes logicielles qu'ils contiennent. Afin d'assurer la qualité de ces systèmes, plusieurs classes logicielles existantes doivent être unitairement testées.

Certaines équipes de développement accordent peu d'importance aux tests unitaires et ont à leur actif plusieurs produits logiciels en cours d'exploitation. Ces systèmes contiennent plusieurs classes logicielles non unitairement testées. Quand ces équipes sont appelées à revoir la qualité de leurs logiciels, suite à une acquisition par une autre compagnie ou à la promotion de leurs produits pour des clients qui exigent des normes de qualité plus élevées, elles sont appelées à développer des suites de tests unitaires pour plusieurs classes de leurs produits logiciels existants.

Des outils d'aide au développement des tests ont vu le jour et permettent, dans le cas du langage orienté objet JAVA (par exemple), d'assister l'écriture des tests unitaires et d'automatiser avec beaucoup de commodité leur exécution. D'autres outils intégrés aux environnements de développement [16] permettent de calculer des métriques mesurant certains aspects du code source orienté objet. L'information que procurent ces métriques permet de mieux appréhender les fonctionnalités en cours de développement. Leur analyse, plus approfondie, permet de révéler (entre autres), le niveau de communication et de dépendance entre les classes, la cohésion des fonctionnalités et les données des classes, l'assignation des responsabilités, ainsi que les parties des systèmes les plus critiques.

Finalement, avec l'apparition et le développement du réseau mondial internet ainsi que la philosophie de l'*open source* [17], les développeurs travaillent de plus en plus en communautés dans des espaces physiques séparés où se côtoient des développeurs séniors et juniors avec moins d'expérience. Le partage d'expériences dans les forums sur le réseau internet est devenu une norme et permet aux jeunes développeurs de tirer profit de l'expérience des séniors dans le développement, les tests et les techniques de maintenance.

Dans ce contexte, il est intéressant d'investiguer de nouvelles techniques permettant à terme de partager l'expérience de manière automatique. Nous nous sommes penchés sur le volet test et nous avons, dans cette thèse, exploré l'utilisation des métriques de code source disponibles pour caractériser l'effort de test sous différentes facettes et identifier les métriques logicielles pertinentes en vue de les combiner aux différentes expériences des équipes de développement. Le but étant d'explorer et de proposer des stratégies d'aide pour l'orientation automatiquement des tests unitaires des classes logicielles, pouvant à long terme s'intégrer et s'adapter à l'environnement de développement communautaire.

1.3 Problématique

Les tests unitaires, comme leur nom l'indique, visent à tester les unités séparément. En orienté objet, cela se traduit par l'écriture de classes tests dédiées aux classes logicielles, qui sont considérées comme les unités de base des programmes orientés objet. Avec les différents chemins du flux possibles dans les différentes méthodes d'une classe, les tests unitaires associés, s'ils se veulent effectifs (et exhaustifs) doivent fournir des entrées de données permettant de couvrir tous les scénarios possibles. Le large nombre de classes développées pour supporter un nombre restreint de fonctionnalités dans les contextes précédents fait que les tests unitaires deviennent vite un important gouffre financier consommant temps, ressources humaines et matériels dédiés au développement du logiciel. Combinés à la taille restreinte des équipes de développement, les tests

unitaires exhaustifs deviennent tout simplement irréalistes dans la plupart des projets logiciels importants, plus encore dans ceux de grande envergure.

Les responsables des tests ou développeurs sont appelés à faire des choix sur les composants à tester et sur le niveau de couverture (des différents chemins de flux) de ces composants. Le choix des classes logicielles pour lesquelles il faudra développer des classes tests lors des tests unitaires (orientation) constitue la problématique principale de notre thèse. Une approche réaliste consiste d'abord à quantifier l'effort de test (du moins quelques-unes de ses facettes) et à identifier les caractéristiques des classes logicielles exigeant des valeurs élevées de cet effort. L'évaluation (de l'effort) et la caractérisation (des classes logicielles) permettent, en effet, de prêter plus attention aux caractéristiques qui rendent les classes logicielles difficilement testables en augmentant l'effort consenti dans les tests unitaires.

La testabilité, la facilité avec laquelle on peut tester une classe logicielle, peut être vue sous l'angle de l'effort que ces classes logicielles requièrent pour leur test. Il s'agit, d'une manière générale, d'un attribut externe et d'une notion complexe faisant intervenir plusieurs facteurs. Cet attribut de la classe logicielle n'est pas directement mesurable par une simple métrique de classe logicielle. Cependant, l'utilisation de combinaisons de métriques logicielles peut aider à appréhender les facteurs du code source (parmi d'autres facteurs) qui influencent la testabilité. Malgré de multiples investigations de la communauté des chercheurs, des questions demeurent notamment sur les métriques de code source liées à la testabilité et le choix des classes logicielles les plus à risque à tester unitairement en priorité. Nous avons, dans cette thèse, investigué les liens entre les métriques de code source et la testabilité de manière générale, évalué l'effort de test à travers une démarche empirique faisant intervenir des systèmes logiciels *open source* de grande taille, avant de proposer une approche basée sur le risque, calculé à partir des métriques logicielles et permettant d'orienter l'effort de test.

1.4 Objectifs

L'objectif principal de cette thèse est de construire des modèles d'aide à l'orientation de l'effort global des tests unitaires vers les composants logiciels les plus à risque. Pour atteindre cet objectif, nous nous sommes fixé des objectifs intermédiaires (spécifiques) afin de diviser la problématique énoncée précédemment et de l'organiser en plusieurs sous problématiques. Le but étant de comprendre, mesurer et caractériser la testabilité unitaire des classes logicielles avant de proposer des approches d'orientation des tests unitaires.

- Évaluation de la testabilité: En nous basant uniquement sur le code du logiciel (un artéfact mesurable disponible), nous avons évalué la testabilité des classes logicielles sous l'angle de l'effort d'écriture et de construction des tests unitaires. Nous avons utilisé des métriques de classes tests unitaires définies dans la littérature que nous avons complétées par deux autres métriques capturant des caractéristiques importantes du test, non capturées par les métriques existantes. L'établissement de lien entre les métriques logicielles et l'effort de test unitaire (effort d'écriture et de constructions des cas de tests unitaires), particulièrement les métriques synthétiques, ouvre des possibilités d'utilisation de ces métriques pour orienter les tests. L'objectif était d'identifier les métriques logicielles les plus liées à la testabilité unitaire des classes logicielles, de confirmer (et compléter) les résultats présentés dans la littérature sur un plus grand nombre de systèmes, mais aussi de confirmer la capacité des indicateurs de qualité, métrique synthétique que nous avons introduite (travaux antérieurs), à capturer la testabilité sous l'angle de l'effort d'écriture et de construction des cas de tests unitaires.

- Évaluation des métriques de test: Les différentes métriques de tests unitaires proposées dans la littérature, ainsi que celles que nous avons introduites pour les compléter, capturent théoriquement des attributs différents des classes de tests unitaires. Nous avons analysé en profondeur ces différentes métriques pour étudier leur pertinence, leur volatilité, ainsi que les dimensions sous-jacentes qu'elles capturent. L'objectif étant de proposer une suite de métriques pertinentes pour les classes de tests unitaires.

- Prédiction des niveaux d'effort de test: Hiérarchiser les classes logicielles en différents niveaux en fonction de l'effort de test unitaire qu'elles requièrent est une étape très importante dans l'orientation de l'effort de test. Elle permet aux responsables de projets de mieux planifier leurs ressources durant les phases de test afin d'apporter les prises en charge particulières (nécessaires) à certaines classes logicielles.

- Orientation de l'effort: Nous avons utilisé des méthodes d'apprentissage automatique en combinant quelques métriques logicielles et les données sur l'historique des classes unitairement testées pour orienter les tests. Nous avons aussi proposé une approche basée sur le risque utilisant uniquement des métriques logicielles dans le but d'orienter l'effort de test unitaire. L'approche développée nous a permis d'évaluer objectivement le risque associé à une classe logicielle sous un certain angle, de mieux comprendre l'effet des facteurs de complexité et de couplage (attributs importants des classes OO) sur ce risque et de proposer une approche d'orientation des tests unitaires basée sur cette analyse. La stratégie d'orientation tient compte des facteurs du code source liés aux probabilités de fautes et à leurs impacts (effet de propagation) sur le logiciel.

1.5 Organisation de la thèse

Nous commençons cette thèse par la présentation de l'état de l'art sur l'analyse du risque, la prédiction des fautes, de leur impact et sur la priorisation des tests dans le chapitre 2.

Après la description, dans le chapitre 3, des différents systèmes utilisés ainsi que leurs domaines d'applications, nous présentons les métriques logicielles que nous avons utilisées dans nos démarches empiriques, les fouilles, ainsi que les techniques d'appariement et d'analyse des données collectées. Nous exposons aussi les outils ayant permis d'effectuer les fouilles, les analyses et les calculs des métriques. Nous présentons, ensuite, la métrique synthétique appelée *indicateur de qualité*, que nous avons introduite dans nos travaux antérieurs à cette thèse, dans le chapitre 4 en présentant les concepts qui la sous-tendent, son formalisme et sa méthode de calcul.

Nos études empiriques débutent dans le chapitre 5, dans lequel nous explorons les liens entre l'effort de test et les métriques logicielles, puis entre l'effort de test et les indicateurs de qualité. Suite à l'exploration de différentes métriques d'évaluation des tests unitaires dans ces analyses, nous avons étudié en détail les propriétés de ces métriques dans le chapitre 6. Nous avons ensuite affiné nos recherches en tentant de prédire les niveaux d'effort de test à l'aide des métriques logicielles dans le chapitre 7.

Après avoir établi le lien entre les métriques logicielles, les indicateurs de qualité et l'effort de test, nous nous sommes intéressés dans le chapitre 8 à l'utilisation de ces métriques dans l'orientation de l'effort de test à travers deux approches différentes. Dans le chapitre 9, nous introduisons une technique d'orientation de l'effort de test basée sur le risque évalué à partir de métriques de code source. Cette dernière nous a permis de mettre en place une démarche empirique d'évaluation de l'efficacité de cette technique d'orientation.

Nous concluons finalement cette thèse en mettant l'emphase sur les résultats obtenus et nous présentons quelques pistes permettant l'exploitation des résultats de cette recherche dans le cadre du développement logiciel pour aider à orienter l'effort de test unitaire.

CHAPITRE 2. ÉTAT DE L'ART

L'objet de cette thèse fait intervenir plusieurs domaines de recherche en génie logiciel. Nous décrivons sommairement, dans les sections qui suivent, l'état des recherches et des principales publications qui sont connexes à notre sujet. Nous montrerons les liens et les différences entre ces travaux de recherche et nous situerons l'objet de nos investigations.

2.1 Analyse du risque dans le processus de développement logiciel

Comme partie importante de la gestion de projet, l'analyse du risque est très utilisée dans les projets en ingénierie. En génie logiciel, plusieurs stratégies de gestion de risque (SERIM - Software Engineering RIsk Model [18], SRE - Software Risk Evaluation [19], SRAM - Software Risk Assessment [20], etc.) ont été proposées par des chercheurs, des structures (SEI - Software Engineering Institute) et des compagnies privées. Il s'agit de processus d'identification, d'évaluation et de réduction du risque dans le cycle de développement du logiciel.

En 2006, IEEE spécifie dans son standard 16085-2006 [21] les lignes directrices permettant de gérer le risque durant le cycle de vie du logiciel. Le standard définit 6 étapes principales à effectuer dans un processus continu: (1) planifier et implémenter la gestion du risque, (2) gérer le profil du risque du projet, (3) analyser le risque, (4) surveiller le risque, (5) traiter le risque, et (6) évaluer le processus de gestion du risque. Les activités doivent être réalisées au début du projet et doivent être répétées à chaque évolution notable des informations d'entrée du projet.

D'un autre côté, le standard ISO 27001: 2005 [22] aborde la gestion du risque dans un projet logiciel selon différentes autres perspectives, qui sont principalement: les technologies de l'information, la sécurité technique et la gestion de la sécurité de l'information et des systèmes. Ce dernier standard met l'emphase sur la sécurisation des systèmes d'information dans un cadre organisationnel. Il guide les organisations vers l'identification et la mise en place d'une politique systématique de gestion du risque.

Parallèlement à ces standards, plusieurs études se sont intéressées à la thématique d'évaluation du risque, de son efficacité et de son applicabilité du point de vue de la gestion de projet de développement de logiciels.

À cet égard, Kontio et Basili [23] ont décrit les trois obstacles majeurs qui freinent l'utilisation systématique des techniques de gestion du risque dans les projets logiciels. Il s'agit, selon eux, de l'ignorance de l'existence de méthodes rigoureuses de gestion du risque, des limitations des méthodes de gestion du risque existantes, et du manque de preuves formelles sur leur efficacité. Pour lever ces trois contraintes, ils proposent la méthode Riskit. Selon les auteurs, cette nouvelle approche est systématique, appropriée et utilisable dans le cadre de la gestion de projet de développement logiciel. Riskit est basée sur un formalisme graphique pour supporter l'analyse qualitative des scénarios de risque avant leur quantification (par la théorie de l'utilité [24]). Le formalisme graphique exposé par les auteurs est centré autour d'un nœud principal représentant l'évènement à risque qui est influencé par de potentiels facteurs de risque. Ces facteurs sont des caractéristiques décrivant l'environnement, pouvant affecter la probabilité de l'évènement. Un évènement à risque peut, selon eux, initier une réponse du gestionnaire, réponse qui réduit la probabilité de survenance de l'évènement. L'impact de l'évènement est par la suite quantifié, ce qui permet au gestionnaire d'en apprécier l'intérêt (définir les traitements appropriés). La gestion du risque est abordée dans ce travail en termes d'objectifs et de priorisation des risques (à traiter) pouvant entrainer l'échec d'un projet logiciel.

Willams et al. présentent dans [19] la méthode d'évaluation du risque pour le développement logiciel: SRE (Software Risk Evaluation). Il s'agit d'un outil de diagnostic et de prise de décision éprouvé. La méthode est notamment utilisée, dans les projets logiciels, par différents secteurs stratégiques du gouvernement fédéral des États-Unis (Department of defense, marine, etc.). La méthode consiste, pour un risque donné, à l'identifier, l'analyser, le catégoriser, planifier sa gestion, agir pour le résorber, et le suivre pour contrôler son évolution. L'approche se focalise plus sur la prévention et la formation des acteurs de projets logiciels sur les notions et sur l'identification des risques. Le rapport

aborde aussi le risque sur le plan de gestion de projet logiciel, et ne considère pas les attributs issus des caractéristiques mêmes du produit logiciel en cours de développement (taille, complexité des spécifications).

Constatant les insuffisances des approches de gestion du risque basées sur des modèles statistiques, utilisées dans la gestion de projets logiciels, Fenton et Neil proposent dans [25] une nouvelle stratégie causale s'appuyant sur les réseaux bayésiens et les métriques logicielles, pour gérer les risques associés à un projet de développement logiciel. Les modèles bayésiens ayant l'avantage de prendre en compte l'incertain qui, selon eux, caractérise fondamentalement les projets de développement. Ils construisent un réseau causal exhaustif ayant comme principaux nœuds la taille, le besoin en ressources, l'appropriation de ressources, l'exactitude de l'implémentation, et les lignes de code. En introduisant par simulation des probabilités a priori sous forme de contraintes sur certains nœuds, ils parviennent à estimer la taille de l'équipe nécessaire au développement de l'application, ainsi que le nombre de fautes et la probabilité de réussite (finalisation) ou d'échec (abandon) du projet. Cependant, aucune étude empirique sur un projet concret ne vient corroborer les conclusions.

Le Modèle SERIM (Software Engineering Risk Index Management) proposé par Karolak dans [18] a pour objectif de minimiser le risque en optimisant les stratégies pour le résorber. SERIM se focalise sur trois éléments de risque: (1) le risque lié à la technique, (2) le risque lié au budget, et (3) le risque lié aux délais. Les trois éléments sont interconnectés par 81 facteurs de risque, qui sont influencés à leur tour par différentes composantes telles que l'organisation, l'estimation, la surveillance, la méthodologie de développement, les outils utilisés, le risque, et l'utilisabilité. Chaque facteur de risque est associé à une métrique de risque spécifique et une question. L'ensemble des questions forme une liste de contrôle qui identifie les risques. Une fois remplies par les responsables de projet, et converties en valeurs numériques par le biais des métriques, les réponses forment un réseau d'informations avec leurs pondérations respectives. Le réseau est finalement utilisé, en combinaison avec un graphe de probabilités, pour calculer les facteurs de risque. Ce modèle ne tient pas compte de certains facteurs comme la complexité du logiciel et les

spécifications. Par ailleurs, le modèle n'a pas été validé empiriquement. Cependant, une implémentation sous forme logicielle a été développée en 1998.

Gupta et Mohd proposent dans [26] le modèle SRAEM (Software Risk Assessment and Estimation Model) permettant d'évaluer, de manière incrémentale, le risque lié à un projet logiciel en fonction des modifications introduites lors de la maintenance du produit. Le SRAEM est basé sur les métriques MCRSRM (Mission Critical, Requirements Stability Risk Metrics) [27] pour évaluer les changements, et sur les modèles d'estimation de la taille des logiciels à partir de leurs spécifications. L'approche SRAEM s'articule autour de trois activités cycliques qui sont: l'identification du risque, l'évaluation du risque et la priorisation du risque. Les auteurs n'ont pas effectué d'études empiriques permettant d'évaluer les performances de leur modèle.

Avec la multiplication des approches, des méta-études ont aussi été conduites afin de déterminer les indicateurs de risque de gestion de projet les plus utilisés dans la recherche en génie logiciel. Menezes et al. [28] ont utilisé une technique de cartographie pour effectuer une analyse systématique de trois grands dépôts de publications scientifiques (Scopus, IEEExplore et Engineering Village). L'étude a été menée sur un ensemble de 506 publications réduites à 63 après filtrage des articles traitant spécifiquement de la gestion du risque dans le domaine du génie logiciel, entre 2004 et 2008. Elle met en lumière 11 indicateurs, considérés par la communauté des chercheurs comme pouvant influencer significativement le risque dans un projet logiciel. Les occurrences de ces indicateurs dans les articles analysés se répartissent ainsi:

- Les caractéristiques de l'équipe de développement (sa taille, sa connaissance de la technologie, sa formation et son expérience) sont relevées dans 24 % des publications.
- Le niveau de maturité de la compagnie revient dans 11 % des publications.
- Le calcul du risque par la méthode de Bernoulli (produit de la probabilité d'un évènement par sa gravité) est recensé dans 11 % des publications.

- Les changements intervenant dans les spécifications, l'implémentation et le produit final apparaissent dans 10 % des publications.
- Le coût financier du développement du produit revient dans 9 % des publications.
- L'exigence de qualité du produit final ainsi que l'effort de test qu'elle entraine sont relevés dans 7 % des publications.
- Les délais de développement reviennent dans 7 % des publications.
- Le code source et ses attributs sont présents dans 7 % des publications.
- Le design et le degré de personnalisation du logiciel sont considérés dans 5 % des publications.
- La complexité du produit fini est citée dans 5 % des publications.
- Et finalement, la taille du projet et du logiciel revient dans 5 % des publications.

Les techniques de gestion et de réduction du risque citées plus haut nous montrent le foisonnement de la recherche dans le domaine de la gestion du risque du point de vue de la gestion de projets logiciels. L'approche de gestion du risque que nous présentons considère le risque à un niveau de granularité plus fin: le composant, et plus précisément la classe dans le paradigme orienté objet. Le risque y est évalué comme résultant de la probabilité et de l'impact d'une faute dans une classe logicielle. Nous utiliserons par la suite cette approche pour prioriser les composants logiciels à tester.

2.2 Prédiction des fautes logicielles

Le Standard Computer Dictionary [29] a défini une faute comme étant une étape, un traitement, ou une définition de donnée incorrecte dans un programme. La défaillance représente l'incapacité d'un logiciel ou d'un composant d'un système logiciel à réaliser ses spécifications avec les performances requises. La défaillance peut être due au matériel ou au logiciel. La faute dans un programme devient une défaillance effective quand elle est rencontrée lors de l'exécution du programme. Ainsi, par exemple, le nom d'un fichier mal orthographié dans un code source est une faute. Le logiciel ne pouvant pas ouvrir le fichier contiendra une erreur, ce qui peut éventuellement conduire à sa défaillance lors de la sauvegarde. Deux constats, entre autres, ont emmené la communauté des chercheurs à s'intéresser de plus près à la prédiction de fautes: (1) Le coût qu'elles engendrent: sur ce point, le NIST (National Institute of Standards and Technology) [9] a estimé entre 22.2 et 59.5 milliards de dollars les pertes nettes occasionnées par les défaillances logicielles aux USA en 2002. (2) La possibilité de réduire leur impact: sur ce point, des études [30,31] montrent que la découverte précoce des fautes dans le processus de développement permet de réduire les coûts liés à leur correction. La prédiction des fautes du logiciel va être, dès lors, un enjeu économique important et emmènera plusieurs chercheurs à s'intéresser au sujet. L'objectif est de déceler les composants les plus enclins à contenir des fautes, très tôt dans le cycle de vie du logiciel. Un composant susceptible de contenir des fautes se voit alors apporter un soin particulier lors des tests, ou encore restructuré en composants moins à risque de fautes. Les techniques de détection sont souvent construites sur des modèles probabilistes, statistiques ou d'intelligence artificielle à partir des données de métriques logicielles. En effet, les corrélations de nombreuses métriques logicielles avec les fautes ont été largement confirmées dans la littérature [32-39].

C'est en analysant huit systèmes logiciels de taille moyenne développés en C++ que Basili et al. ont montré empiriquement dans [36], grâce à la régression logistique univariée et multivariée, que les métriques de Chidamber et Kemerer (CK, ensemble de métriques de code source définies pour les systèmes orientés objet) [37] sont particulièrement corrélées aux fautes. Dans la même veine, Briand et al. [38] ont démontré que dans les

classes logicielles des systèmes orientés objet, la taille, la fréquence des invocations des méthodes et la profondeur de l'arbre d'héritage étaient fortement corrélées au nombre de fautes détectées dans un composant écrit en C++. Plus tard, El Emam et Melo [39] confirmeront empiriquement ces résultats en mettant en évidence le lien entre les métriques de design objet et la probabilité de présence de fautes dans les classes logicielles. En investiguant des moyens de prédiction toujours plus précoces, les chercheurs se sont intéressés aux liens entre les métriques de spécifications et les fautes. Dans ce cadre, Kaur et al. [40] combinent un groupe de métriques de spécifications, métriques issues des spécifications logicielles et déterminées beaucoup plus tôt dans le cycle de développement, à un groupe de métriques de code source collectées plus tard dans le cycle. Ils montrent qu'il est possible de prédire assez tôt, grâce aux métriques issues des spécifications, les classes sujettes aux fautes et qu'en plus, en combinant ces métriques avec celles obtenues plus tard à partir du code source, on peut nettement améliorer les performances des modèles de prédiction. Les auteurs recourent à la technique du clustering (classification) sur des données issues des dépôts du MDP (Metrics Data Program) [41] et qui concernent des systèmes écrits en C++. Ils produisent un modèle de prédiction plus performant que ceux basés sur l'un ou l'autre des deux groupes de métriques pris séparément.

Aggarwal et al. [42] étendent l'étude de Briand et al. [38] à 12 systèmes écrits en JAVA, dans un environnement contrôlé. Après une analyse par composantes principales qui leur a permis de déceler le recoupement des informations capturées par les métriques, ils construisent des modèles logistiques univariés sur les données et parviennent à une exactitude de prédiction de l'ordre de 90 %. Ils notent, aussi, que les métriques de couplage sont particulièrement corrélées à la probabilité de détection de fautes dans les classes logicielles.

Rathore et al. [43] investiguent de leur côté la capacité de prédiction des modèles construits à partir des métriques orientées objet (associées aux caractéristiques du paradigme orienté objet - POO) tels que le couplage, la cohésion, la complexité, l'héritage et la taille pour prédire les classes susceptibles de contenir des fautes. Les auteurs utilisent

la régression logistique et l'analyse de la courbe ROC (Receiver Operating Characteristic) afin de sélectionner les métriques les plus performantes. Ces dernières servent ensuite de base d'apprentissage pour les algorithmes d'intelligence artificielle. Après avoir testé cette approche sur des données de PROMISE (Predictor Models In Software Engineering) [44], les auteurs concluent que les métriques orientées objet liées au couplage et à la complexité produisent des modèles significativement plus précis (pour la prédiction de fautes) que les autres métriques.

D'autres études font appel aux méthodes d'apprentissage automatique pour construire des prédicteurs de fautes à partir de (l'information sur) l'historique des fautes et/ou des métriques logicielles. Ainsi, Gyimòthy et al. [45] se sont penchés sur le cas des logiciels *open source*. Ils bâtissent des modèles par apprentissage automatique (C4.5 et réseaux de neurones) directement sur les métriques CK [37]. L'analyse empirique est effectuée sur 7 versions de la suite logicielle Mozilla (le navigateur et le mailer). Elle montre, d'une part, qu'il est possible de prédire les classes les plus prédisposées aux fautes à partir de l'historique des attributs orientés objet et, d'autre part, que le couplage et la complexité présentent des résultats particulièrement intéressants. L'analyse de l'évolution de la suite de logiciels qu'ils ont produite par la suite montre que les corrélations entre les fautes et les métriques CK [37] persistent sur plusieurs versions des logiciels *open source*.

Pour Shatnawi [46], démontrer le lien entre les métriques logicielles et les fautes ne suffit pas pour une utilisation effective des approches basées sur les métriques logicielles pour la prédiction des fautes dans le processus de développement. Il entreprend dans [46] de déterminer les seuils de valeurs pour les métriques CK au-delà desquels les classes logicielles seraient plus à risque de présenter des défaillances. Il mène une analyse empirique sur trois systèmes logiciels en s'inspirant d'un modèle épidémiologique [47] d'évaluation de seuils de risque, pour calculer les seuils de valeurs acceptables pour chaque métrique dans le cadre du logiciel. Une analyse de régression logistique lui permet de restreindre l'ensemble CK aux métriques de complexité et de couplage comme étant les plus corrélées à la probabilité de présence de fautes. Après la détermination des seuils sur la version 2.0 d'Eclipse, l'auteur fait une validation sur la version 2.1 en construisant un

prédicteur (sous forme d'arbre de régression) et croise les résultats expérimentaux avec les résultats effectifs. Il démontre, ainsi, la très bonne capacité de prédiction de fautes pour les seuils déterminés plus tôt. À des fins de comparaison, l'auteur détermine les seuils d'autres systèmes (il valide aussi leur capacité de prédiction), et note que les métriques significatives sont les mêmes pour tous les systèmes, mais que les (valeurs de) seuils dépendent fortement du système considéré et ne peuvent donc pas être utilisés de manière absolue.

L'historique sur le dépôt ainsi que les métriques des versions passées d'un logiciel constituent une ressource informationnelle importante pour les modèles de prédiction de fautes. Cependant, cette ressource n'est pas toujours disponible ou accessible pour différentes raisons (première version du logiciel, non-intégration d'un système de gestion de versions lors du développement, difficulté à collecter l'information, etc.). Pour répondre à ce besoin, Catal et al. [48] ont considéré le problème de prédiction de fautes en l'absence d'historique de données du système en cours d'analyse. Les auteurs ont utilisé la classification pour effectuer un apprentissage non supervisé se basant sur les métriques de taille, de complexité, du nombre d'opérandes et du nombre d'opérateurs. Ils ont démontré qu'en utilisant des valeurs de seuils déterminées par la littérature, en analysant plusieurs logiciels et moyennant quelques adaptations, qu'il était possible de parvenir à un taux de prédiction acceptable. Cette étude met en lumière une certaine portabilité des règles issues d'un système vers un autre, qui permet à terme de réutiliser de manière automatique l'expérience des développeurs afin de prédire très tôt les fautes dans le processus de développement.

Shatnawi a proposé plus récemment [49] une technique de dopage (boosting) basée sur le rééchantillonnage pour améliorer les modèles de prédiction de fautes basés sur les métriques logicielles. L'approche tire l'avantage de la singularité des distributions des données de fautes dans les classes au fil du temps (distribution fortement hachée), pour doper trois classificateurs (bayésien naïf, le réseau bayésien et le KNN) construits sur un sous ensemble des métriques de CK. La méthode proposée est appliquée au système Eclipse. À des fins de comparaison, l'auteur applique une autre technique de dopage bien

connue (SMOTE) sur les mêmes données. Les résultats de la nouvelle méthode montrent une nette amélioration de la capacité de prédiction des classificateurs par rapport à SMOTE.

Ces différentes études et bien d'autres ont confirmé le lien de corrélation des métriques logicielles avec les fautes et leur capacité à les prédire. Il ressort aussi que les métriques de couplage et de complexité sont particulièrement liées aux fautes. Ces métriques de niveau du code source présentent un avantage en ce sens qu'elles se calculent de manière automatique et s'intègrent donc facilement à des outils de développement. Cependant, la nécessité de détenir le code source (partiel ou entier du logiciel) retarde un peu leur considération dans le processus de développement et ne permet leur utilisation qu'une fois le code implémenté. Vu leurs très forts liens avec la probabilité de présence de fautes, ces métriques constitueront les piliers des modèles de priorisation et d'analyse du risque que nous présenterons dans ce document.

La prédiction de fautes dans les composants logiciels est importante en ce sens qu'elle permet d'isoler des composants potentiellement à risque de défaillance. Elle permet aux responsables des projets d'apporter les traitements adéquats à ces composants. Cependant, les fautes n'ont pas les mêmes effets sur le logiciel. Certaines n'impactent pas le bon fonctionnement du système ou ne nécessitent que peu d'efforts de mise au point tandis que d'autres peuvent: (1) compromettre le projet en phase de développement en nécessitant beaucoup plus d'efforts entrainant des délais, ou encore (2) rendre le système tout simplement inopérant durant la phase d'exploitation. Le premier point est souvent traité dans le processus de gestion du risque du projet logiciel. Le deuxième point, quant à lui, est lié à la sévérité des fautes se produisant dans les systèmes. Dès lors, il est important pour les responsables de projets, en plus de pouvoir isoler les composants prédisposés aux fautes, de déterminer la gravité des fautes prédites en termes d'impact et de sévérité par rapport au système. Ce constat nous emmène, dans la section suivante, à nous intéresser à la sévérité des fautes.

2.3 Prédiction de l'impact et de la sévérité des fautes

Le lien entre les fautes et les métriques logicielles particulièrement le couplage, la taille et la complexité a largement été mis en évidence par les différentes investigations citées dans la section précédente. Par conséquent, les métriques logicielles sont de plus en plus utilisées dans le processus de gestion de projets de logiciels [50]. Cependant, une autre dimension des fautes, la sévérité, est tout aussi importante pour le responsable de projet. La sévérité représente les différents niveaux d'impacts négatifs qu'une faute peut avoir sur un logiciel lors de son exploitation [51,52].

L'analyse des dépôts publics des logiciels lors de la collecte de données pour cette thèse nous montre que durant les phases de test et d'exploitation, les fautes rapportées sont hiérarchisées selon leurs niveaux de sévérité avant d'être affectées aux développeurs pour correction. Ce processus de hiérarchisation se fait souvent manuellement [53], et à cause de sa complexité et des ressources qu'il requiert [54], ne se fait pas de manière formelle et systématique. En pratique, les responsables de projets font une analyse sommaire des constats faits par les testeurs ou les utilisateurs sur le comportement du système défaillant. Ils se basent, ensuite, sur leur connaissance du système pour attribuer un niveau de sévérité. Il en résulte que certaines fautes ont des impacts sous-estimés sur le système (plus d'impact que prévu sur les sorties, ou nécessitent plus de ressources pour la correction). L'information sur la sévérité des fautes est importante pour les responsables de projets en ce sens qu'elle permet de hiérarchiser les niveaux de sévérité des différentes fautes et par la suite de mieux répartir les ressources durant les processus de test afin de (entre autres) respecter les dates de mise en production du logiciel [53]. Peu d'études se sont penchées sur le lien entre la sévérité des fautes et les métriques logicielles à cause principalement de la difficulté à collecter des données fiables et du travail d'analyse que l'activité exige pour associer sévérités, fautes et classes [55]. Les fautes sont généralement catégorisées en différents niveaux par les responsables de projet selon les ressources nécessaires à leur correction ou leur impact sur la phase d'exploitation (bloquante, mineure, majeure, etc.).

Investiguant sur les critères de sévérité basés sur l'impact en exploitation, Zhou et Leung [51] ont analysé les relations entre les métriques de code (issues en grande partie de la suite de métriques CK) et les fautes en termes de sévérité. Ils ont mené les investigations sur l'historique provenant du dépôt public de données de métriques logicielles de la NASA [41]. L'expérience consiste à utiliser les métriques de design orienté objet (complexité, héritage, couplage, cohésion) et la taille (comme base de comparaison), pour prédire l'occurrence des fautes de différentes sévérités dans les composants. Les auteurs ont ainsi analysé 145 classes logicielles contenant 549 fautes réparties en 5 catégories selon le niveau de sévérité (de bloquant à trivial). Les approches utilisées sont la régression logistique binaire univariée, multivariée, le réseau bayésien naïf, la forêt aléatoire et l'algorithme du K-plus-proches-voisins (KNN). Les résultats confirment, d'une part, que la plupart des métriques orientées objet issues de la suite CK sont liées aux fautes à travers leurs sévérités et que, d'autre part, la capacité des métriques à prédire la présence de fautes dépend du niveau de sévérité considéré. Enfin, Zhou et Leung constatent que les métriques orientées objet prédisent mieux les fautes de faible sévérité que les fautes de grande sévérité. Un autre constat issu des résultats d'analyse de régression est que le couplage est significativement plus lié à la sévérité des fautes que les autres métriques orientées objet. Notons, enfin, que dans cette étude, les auteurs n'ont pas exploré le modèle de régression logistique multinomiale plus adéquat dans ce cas (la variable qualitative dépendante ayant plus de deux états). Ils ont, à la place, regroupé les 5 niveaux de sévérité en deux groupes et ont appliqué tour à tour la régression logistique binaire.

Pour faciliter le travail de hiérarchisation des niveaux de fautes découvertes, Iliev et al. ont cherché dans [53] des moyens automatiques permettant d'assigner la sévérité aux fautes. Leur méthode se base sur l'analyse des spécifications et du design du système. Ils utilisent des techniques d'intelligence artificielle pour l'analyse automatique de texte basée sur le langage OWL-DL (Ontology Web Language Description Logics) afin d'extraire les données et classifier automatiquement les fautes en différents niveaux. L'étude est effectuée sur un logiciel industriel et les résultats sont comparés aux classifications effectuées par des responsables de projets. Dans cette étude, 58 % des fautes ont été

classifiées correctement. En plus de l'automatisation de l'assignation des niveaux de sévérité, leur approche, soulignent-ils, présente l'avantage de tenir compte systématiquement des spécifications du client et donc de ses priorités lors de la hiérarchisation des fautes, contrairement à l'assignation manuelle. Ce qui pourrait aussi expliquer, d'après eux, les résultats faibles obtenus lors de leur expérimentation. Dans le même registre, Sandhu et al. [52] ont mis à contribution les informations de 21 métriques de code source sur lesquelles la classification non supervisée DBSCAN (Density-Based Spatial Clustering of Applications with Noise) et la logique floue (Neuro-Fuzzy) ont été appliquées pour prédire les niveaux de sévérité des fautes pouvant se produire dans un composant logiciel. L'analyse empirique a été effectuée là aussi sur les dépôts publics de la NASA [41] et a conduit à une exactitude de plus de 86 % pour les deux approches.

La prédiction des fautes et de leur sévérité permet une meilleure distribution des ressources nécessaires à leur correction. Nous nous baserons sur les liens et le faisceau d'indices établis par ces quelques investigations (entre les métriques logicielles et la sévérité des fautes) pour bâtir une approche de priorisation des tests. Il reviendra alors aux responsables de projets logiciels, détenant ces informations, d'orienter et de prioriser efficacement les ressources, particulièrement pour les phases de test. Cette activité a vu naitre beaucoup d'approches et de stratégies ayant pour objectif d'améliorer l'efficacité des tests. La priorisation a ainsi fait l'objet d'investigations majeures par la communauté des chercheurs en génie logiciel. L'état de l'art de ces investigations est exposé dans la section suivante.

2.4 Orientation et priorisation des tests et des efforts de test

Le test fait partie du processus de vérification et de validation du logiciel et constitue un élément central dans l'assurance qualité du logiciel. Il représente la révision ultime des spécifications du design et du code source. Il constitue 30 à 40 % de l'effort de développement dédié à un projet logiciel [8] et plus de 50 % du coût total du produit [11]. La prédiction et l'estimation de l'effort de test ont fait l'objet de plusieurs investigations dans la littérature [56-57]. Le but principal de la prédiction étant de déceler, dans le

logiciel, les composants nécessitant un effort relativement important de vérification et de validation afin de permettre aux responsables de projet de mieux (entre autres) planifier leurs ressources.

Le test permet, en général, de déterminer les écarts entre le comportement attendu et le comportement réel d'un logiciel (ou d'un composant du logiciel). Plusieurs types de tests existent et peuvent être regroupés différemment, selon différentes perspectives. Suivant les niveaux d'accessibilité, nous distinguons deux approches de test [8]: (1) les tests qui tiennent compte de la logique de programmation interne du logiciel lors de la création de leurs scénarios, aussi appelés tests en « boite blanche », et (2) les tests qui considèrent d'autres propriétés indépendantes de la logique de la programmation du logiciel. Il s'agit de tests dits en « boite noire ». Suivant le niveau de granularité du logiciel, nous retrouvons (entre autres) les tests unitaires dont l'objectif est de vérifier les composants séparément (unitairement), et les tests d'intégration qui vérifient le bon fonctionnement des composants ensemble. Ces deux types de tests sont particulièrement bien intégrés au processus de développement dans le paradigme orienté objet ainsi que dans les environnements de développement, où plusieurs frameworks permettent de les supporter [59-60]. Selon la phase du cycle de vie du logiciel, on peut distinguer d'un côté, les tests de régression qui peuvent être réalisés en boite noire ou boite blanche et permettent de vérifier que les nouveaux changements (évolution, correction de bogue, etc.) intervenus dans le logiciel n'ont pas introduit des comportements non souhaités (par rapport aux spécifications) et, de l'autre, les tests de validation, réalisés après l'implémentation de certaines fonctionnalités, permettent de vérifier leur conformité avec les spécifications établies. Il existe, finalement, d'autres types de tests permettant de vérifier et de valider différents paramètres tels que la charge, la robustesse, etc.

En prenant en compte la logique de programmation, les techniques en boite blanche peuvent cibler des parties précises dans les composants à tester. Cependant, une couverture de test complète est souvent irréalisable au vu de la taille du logiciel et de la multitude, de scénarios qu'engendre notamment cette technique. À cela viennent s'ajouter les contraintes de délais et de ressources liées au projet. Les responsables de

projet sont obligés de faire des choix sur les composants (à tester), les scénarios (de test) et l'ordre dans lequel ils doivent être effectués [15]. On parle d'orientation et de priorisation des tests. L'orientation des tests cible les composants à tester absolument (ceux qui sont susceptibles de contenir des fautes) et la priorisation vise l'optimisation de l'exécution des tests. Elle indique l'ordre dans lequel doivent s'exécuter les cas tests (pour optimiser une fonction coût).

Au sujet de l'orientation, les recherches se focalisent sur la prédiction des classes prédisposées à contenir des fautes (c.f. section 2.3) sans nécessairement prioriser les composants à tester, tandis que la priorisation s'intéresse en général, à l'ordre d'exécution des cas de test. Sur ce point, différentes stratégies ont fait l'objet de recherches exhaustives dans la littérature. Rothermel et al. [61] ont été les premiers à regrouper, selon le critère à optimiser, les différentes approches de priorisation des cas de test en cinq grandes familles. Cependant, cette approche ne concernait que les tests de régression. Pour tenir compte des autres types de tests, nous nous sommes inspirés de cette classification et de celle plus récente de Jatain et Garima [62] pour regrouper les principales techniques en quatre familles selon les directions de recherche actuelles: (1) La priorisation des tests basée sur l'optimisation du taux de détection de fautes, (2) la priorisation des tests basée sur l'optimisation du taux de couverture, (3) la priorisation des tests basée sur les données d'historique, et (4) la priorisation des tests basée sur l'optimisation du taux de détection de fautes à haut risque. Dans ce qui suit, nous présentons la revue de la littérature selon ces différents angles.

2.4.1 Priorisation basée sur l'optimisation du taux de détection de fautes

Cette famille de stratégies réordonne les cas de test de sorte à détecter le maximum de fautes en premier. L'objectif étant de faire décroitre, au plus vite, la (fonction de) distribution des fautes restantes dans le composant sous test.

Dans ce contexte, Rothermel et al. [61] montrent empiriquement, dans un environnement contrôlé, qu'en changeant l'ordonnancement des tests de régression, il était possible de détecter plus de fautes et plus vite (dès les premiers tests) dans le logiciel. Pour mesurer

les gains effectifs en termes de nombre et de taux de détection des fautes, les auteurs introduisent le taux de fautes détectées par fraction de test exécutée qu'ils nomment APFD (Average Percentage of Faults Detected). Cette métrique sera ensuite utilisée par l'ensemble de la communauté des chercheurs pour comparer les performances des différentes stratégies proposées. L'étude menée dans [61] compare plusieurs stratégies d'optimisation parmi lesquelles la priorisation optimale, qui consiste à maximiser le taux de fautes découvertes en ciblant les cas de test qui exposent les parties contenant des fautes. Ce qui n'est possible évidemment que dans un environnement contrôlé. En pratique, cette stratégie est irréaliste (les classes fautives n'étant pas connues d'avance), mais expérimentalement, la technique pressentie pour donner les meilleurs résultats dans l'article, servira de base de comparaison avec les autres stratégies utilisées. Les autres intègrent dans leurs études (entre autres) deux techniques: (1) le total FEP (Fault Exposing Probabiliy) maximisant la probabilité totale de découverte de fautes prédites par des modèles, et (2) l'incrémental FEP qui, de manière itérative, sélectionne les cas de test qui maximisent le facteur FEP. Les résultats montrent que ces deux techniques sont nettement meilleures en termes d'APFD que l'absence de priorisation ou la priorisation aléatoire. Elles restent, cependant, moins performantes que la priorisation optimale.

Malishevsky et al. noteront plus tard [63], que la métrique APFD ignore non seulement des dimensions importantes liées aux fautes (coûts différents, impacts différents sur le logiciel), mais aussi les différences dans les cas de test (taille et effort requis entre autres). Ils étendent la métrique APFD dans [61] pour tenir compte des coûts entrainés par les fautes et les différences dans les cas de test. Ils montrent empiriquement que cette deuxième métrique (nommée $APFD_c$) reflétait mieux les faits et donnait pour les techniques utilisées dans le papier [53] une meilleure priorisation en termes de performance selon $APFD_c$.

Yu et Laub [64] proposent une priorisation basée sur des approches théoriques de la prédiction de fautes ainsi que leurs relations avec les cas de test. Les auteurs appliquent leur stratégie aux tests de régression suivant les changements dans l'implémentation de quelques expressions logiques. En plus de la métrique APFD, la métrique FATE (Fault

Adequate Test set sizE) est utilisée pour évaluer le gain effectif de leur approche. Ils parviennent à réduire, dans le meilleur des cas, à 72 % la fraction de test nécessaire pour détecter l'ensemble des fautes.

Carlson et al. [65] ont utilisé le clustering (méthode hiérarchique agglomérante) sur des données issues de la complexité cyclomatique de l'historique du taux de couverture (des tests) ainsi que des fautes logicielles, pour optimiser le taux de détection de fautes lors des tests. Ces données ont été utilisées seules ou en combinaison entre elles. La stratégie a été appliquée à un système industriel de grande taille (700K lignes de code réparties sur 827 classes) sur trois versions. Les résultats montrent une nette amélioration du taux de prédiction de fautes au sens APFD par rapport à la méthode (sans priorisation formelle) utilisée par le contrôle technique de l'industriel.

Vu la multiplicité des techniques de priorisation basées sur l'optimisation de la détection de fautes développées par les chercheurs, Elbaum et al. [66] conçoivent une méta-stratégie de sélection de la technique la plus efficace en fonction du type de logiciel en cours de développement. Pour cela, ils appliquent un ensemble de quatre techniques de priorisation aux huit mêmes programmes et analysent les valeurs de la métrique APFD. Ils constatent que, dépendamment des spécificités des programmes, les performances des stratégies variaient. Ils réalisent alors une méta-classification (utilisant un arbre de classification) au niveau des différentes stratégies de priorisation des tests et des jeux de données issus d'un programme, pour déterminer la meilleure technique de priorisation pour ce programme. Le résultat des analyses donne des indices intéressants sur les types de techniques de priorisation appropriées ou non à un logiciel donné, selon ses spécificités.

2.4.2 Priorisation de tests basée sur l'optimisation du taux de couverture

Dans cette famille d'approches, l'objectif premier est de maximiser le taux de couverture des tests sur certains artéfacts qui peuvent subir des modifications comme les lignes de code, les branches de code, les contrôles, les changements, les classes ou encore les spécifications (dans ce sens, nous incluons dans cette catégorie les techniques de

priorisation basées sur les spécifications du client). Il découle souvent de cette technique une amélioration du taux de détection de fautes, bien que ce corollaire ne constitue pas l'objectif principal de cette famille de stratégies. Plusieurs auteurs [61,67-72] ont investigué ce domaine et ont développé différentes techniques basées sur des modèles statistiques, probabilistes et d'intelligence artificielle (algorithmes d'apprentissage/ génétiques).

Pour optimiser les tests de régression, Wong et al. [72] suggèrent une des premières stratégies de priorisation d'une suite de tests de régression. Elle optimise le facteur « coût par taux de couverture (des branches ayant subies le changement) ». Les auteurs se focalisent sur la priorisation d'un sous-ensemble de cas de test d'une suite de tests par la technique du « safe regression selection ». Les cas de test qui atteignent les zones modifiées sont sélectionnés et placés en tête de file, les autres cas sont placés à la suite dans l'ordre d'exécution. L'objectif est d'optimiser le rapport entre le coût et le taux de couverture et vise globalement à couvrir les zones ayant subies des changements. Cette technique est un simple algorithme gourmand et itératif qui cherche à couvrir le plus de branches de code ayant subies le plus de changements.

Des algorithmes plus complexes seront considérés plus tard dans les approches de couverture. C'est ainsi que Mirarab et al. [67] utilisent le réseau bayésien pour construire un modèle unifié à partir des informations fournies par les métriques CK, les changements, et les taux de couverture. Ils intègreront plus tard, un mécanisme de retour d'informations dans le processus d'apprentissage du modèle bayésien (la rétroaction) [73] pour en améliorer les performances. Le modèle de priorisation, ainsi construit, optimise le taux de couverture et comme corollaire, améliore le taux de détection de fautes au sens de la métrique APFD. La version sans rétroaction est appliquée d'abord sur huit systèmes et les auteurs notent une amélioration significative du taux de détection par rapport à l'absence de priorisation. Le modèle présenté dans [73], intégrant la rétroaction dans le réseau bayésien, a été validé sur cinq systèmes logiciels *open source* et a conduit à une amélioration relative des résultats par rapport à la version sans rétroaction dans [67].

Dans la comparaison des stratégies de priorisation [61], Rothermel et al. ont utilisé plusieurs techniques de priorisation comme mentionnées plus haut. Parmi lesquelles, quatre techniques ciblent le taux de couverture à différents niveaux du programme: (1) la couverture totale des branches est une technique de priorisation qui consiste à trier les cas de test (après instrumentation du code) dans un ordre qui maximise le nombre total de branches couvertes, (2) la couverture incrémentale des branches qui consiste à chaque itération du tri, à sélectionner le cas de test qui couvre une nouvelle branche du flux, (3) la couverture totale des contrôles, et (4) la couverture incrémentale des contrôles. Les deux dernières techniques sont similaires aux deux précédentes stratégies à la différence qu'elles ciblent les contrôles et non les branches dans le programme. Après avoir été appliquées sur huit systèmes, les quatre techniques démontrent des performances similaires aux techniques ciblant directement l'optimisation du taux de fautes.

Toujours dans la priorisation des tests de régression, Walcott et Kapfhammer [71] s'intéressent à la contrainte de temps dans les techniques de priorisation. Ils présentent une stratégie de priorisation basée sur les algorithmes génétiques avec des contraintes temporelles. La fonction fitness de l'algorithme génétique et le taux de couverture (considéré dans deux différents niveaux de granularité: le niveau bloc de code et le niveau méthode). Pour les deux systèmes logiciels étudiés empiriquement, les résultats montrent un meilleur taux de détection (APFD) des algorithmes qui considèrent la granularité de niveau bloc de code. L'étude montre, aussi, la possibilité de prendre en compte explicitement la contrainte temporelle lors de la priorisation.

Dans le cadre des tests fonctionnels, Bryce et Memon [70] proposent une stratégie de priorisation des suites de test qui vise à maximiser la couverture de l'ensemble des interactions possibles au niveau de l'interface utilisateur. Après avoir passé en revue toutes les combinaisons de scénarios de test possibles (permutations), grâce aux spécifications et à l'analyse de l'interface utilisateur, leur algorithme (de type gourmand « gready ») va maximiser le taux de couverture avec le moins de permutations. À des fins de comparaison, différents critères sont introduits dans cet algorithme en plus du taux de couverture. Finalement, les auteurs appliquent la stratégie à quatre systèmes et

déterminent les performances des techniques grâce à la métrique APFD. Les résultats montrent que cibler la couverture en se basant sur leur technique donne le meilleur taux de détection (APFD) de fautes.

Certains auteurs utilisent le taux de couverture pour minimiser (réduire) la taille d'une suite de test. La réduction des suites est souvent basée sur la couverture [68]. Il s'agit de trouver un sous-ensemble réduit de la suite de tests originale qui maximise le taux de couverture (code/composants/spécifications). Notons que, d'une part, l'utilisation itérative de la minimisation des cas de test (en sélectionnant les cas de test retenus à chaque étape) permet de prioriser les cas de test et que, d'autre part, l'utilisation de seuils pour la métrique APFD dans la priorisation (tronquer le résultat à partir d'une certaine valeur satisfaisante de la métrique coût APFD) permet de réduire (minimiser) la suite de tests. Les stratégies mises en place par différents chercheurs pour réduire ou prioriser les tests sont équivalentes dans le sens où chaque type de stratégie peut permettre à la fois de minimiser et de prioriser les suites de test.

Convaincus du lien entre la minimisation (réduction) et la priorisation (ordonnancement), Leon et Pdgurski [69] présentent une comparaison empirique de quatre différentes techniques de réduction pour les suites de test de grande taille: réduction de suites de test, priorisation par couverture incrémentale, filtrage par clustering à un échantillon et échantillonnage par fautes. Les deux premières sont basées sur la maximisation de la couverture de test, les deux dernières sur l'analyse du profil opérationnel des tests avec des mesures de dissimilarité des cas de test. Leurs résultats indiquent que pour cibler les fautes, leurs techniques (basées sur la dissimilarité) pouvaient être aussi efficaces sinon plus que les techniques basées sur les taux de couverture, mais aussi que les deux techniques sont complémentaires en ce sens qu'elles ciblent des fautes différentes. Ils montrent, alors, que les techniques combinées donnent de meilleurs résultats que les techniques basées sur le taux de couverture prises séparément.

2.4.3 Priorisation basée sur l'historique

Les techniques précédentes n'ont que peu ou pas de mémoire vis-à-vis de l'historique des données du logiciel. Les techniques que nous allons présenter dans cette section se basent principalement sur l'historique de l'information du logiciel (issu des dépôts logiciels). Parmi les données analysées dans ces techniques figurent l'historique des fautes, l'historique des suites de test, l'historique des changements du code source, etc. Les données sont par la suite analysées de manière automatique ou manuelle et serviront de base informationnelle à des algorithmes qui trieront les suites de test dans l'ordre approprié. Dans cette famille, il n'existe souvent pas de fonctions coût explicites à optimiser, mais la métrique APFD s'améliore comme conséquence de l'application de ces techniques de hiérarchisation.

Dans la recherche de modèles à mémoire, Kim et Porter [74] mettent à profit l'historique de l'exécution des données pour prioriser les cas de test dans une suite de tests de régression. Les auteurs investiguent les performances à long terme des techniques de priorisation en se basant sur l'historique des données, le tout dans un environnement sous contraintes (de temps et de coût). Dans leur approche, la probabilité de sélection d'un cas de test est une fonction convexe des probabilités précédentes et certaines caractéristiques qui dépendent du modèle (découverte ou non de fautes, couverture des branches, des fonctions, des contrôles, etc.) et des résultats de la précédente exécution du cas de test. Ils ont comparé leur modèle à différents modèles (sans mémoire) ainsi qu'à la technique aléatoire notamment. Les résultats montrent, entre autres, que l'utilisation des historiques de données peut réduire les coûts et améliorer l'efficacité des tests de régression à long terme.

Dans les tests en boite noire, Qu et al. [75] présentent un processus général de priorisation des cas de test de régression, lorsque le code source n'est pas disponible. Les auteurs ont implémenté un algorithme basé sur l'historique des tests et l'information sur leur exécution. L'idée de base consiste à regrouper les cas de test (réutilisés) en fonction du type et du nombre de fautes qu'ils ont révélés dans leurs exécutions précédentes, pour

ensuite prioriser et réajuster la sélection en fonction des fautes révélées dans l'exécution courante. Après avoir introduit deux métriques d'évaluation de performances inspirées de APFD et spécifiques aux tests en boite noire, les auteurs mènent l'expérimentation sur deux systèmes logiciels et évaluent l'efficacité de leur algorithme en le comparant aux performances du modèle aléatoire. Les résultats montrent que l'algorithme peut aider à améliorer les performances de test.

Plus récemment, Lin et al. [76] ont investigué sur les aprioris des modèles basés sur l'historique des données de test et sur l'importance relative accordée à la mémoire de l'information d'une version à l'autre. Par exemple: (1) les modèles qui utilisent l'information sur les cas de test supposent que ceux-ci n'ont pas varié d'une version à l'autre ou que leur variation n'affecte pas de manière significative leur cote (lors de la sélection pour la priorisation). (2) Le modèle de Qu et al [75] suppose que les résultats d'un cas de test exécuté sur une version du logiciel immédiatement précédent ont la même importance sur la cote du même cas de test lorsqu'exécuté sur les versions suivantes. Les auteurs proposent un modèle relativisant les effets de ces informations en considérant les informations issues de l'historique des fautes et du code source. Lin et al. étendent ainsi les investigations de Liu et al [77] qui ont déjà démontré l'importance relative des autres informations issues de l'historique du logiciel dans la priorisation des cas de test. Une expérimentation empirique est menée sur sept programmes issus du dépôt de données SIR (Software-artifact Infrastructure Repository) [78] afin de déterminer l'importance relative de ces variations et l'apport de la nouvelle approche proposée. Les résultats invalident les suppositions des modèles basés sur l'historique et indiquent que l'approche combinant l'information sur l'historique de fautes et de code source donnait une meilleure valeur de $APFD_c$ (extension de la métrique APFD proposée par Malishevsky dans [63]).

Les stratégies basées sur l'historique des études n'investiguent pas l'utilisation de l'information issue de projets différents. En effet, l'historique des tests est appliqué à d'autres versions du même logiciel. L'information retenue dans cet historique est fortement liée aux spécificités du logiciel en cours de test, à savoir l'historique des cas de

test. L'approche que nous présentons dans cette thèse s'inspire de cette famille de stratégies. Cependant, nous investiguons la possibilité d'utiliser des connaissances acquises à partir de projets différents et ciblons les classes logicielles à tester (et non les suites de test) pour orienter l'effort de test et prioriser les composants dans un processus de test unitaire et d'intégration par exemple.

2.4.4 Priorisation basée sur le risque

Le concept de priorisation basée sur le risque ou RBT (Risk Based Testing) a été introduit par Bach R. dans [79]. L'approche vise à prioriser les cas de test qui couvrent les composants à haut risque en premier pour orienter les tests. Les risques sont évalués par différents mécanismes (théoriques ou empiriques) en termes de probabilités et d'impacts de fautes. Différentes recherches traitent cette approche dans la littérature [80-83] et ont produit des modèles qui améliorent la capacité de détection de fautes lors des tests.

Partant du postulat que le test est motivé avant tout par le risque, Bach propose dans [80] une heuristique permettant de déceler et de trier le risque de défaillance selon sa probabilité et son impact. L'approche est basée sur une liste de vérification de facteurs de risque internes, externes, mais aussi des évènements risqués ainsi que leur impact. Tous ces éléments doivent être précisés et ajustés tout le long du cycle de vie du projet logiciel pour finir par épouser les spécificités de produit.

Suite au modèle proposé par Bach, l'évaluation quantitative du risque a très vite intéressé les chercheurs. Amland définit dans [81] une suite de métriques et de fonctions coût pour supporter les tests basés sur le risque. Il détermine, ensuite, une technique simple de choix d'une de ces fonctions. E. Souza et al. utilisent un modèle GQM dans [69] pour définir une suite de métriques supportant la priorisation des tests basée sur le risque. Ils déterminent une approche (RBTProcess) intégrant les différentes métriques dans les six étapes classiques de gestion du risque définies par IEEE16085:2006 [21]. Dans cette suite, les métriques sont groupées en trois familles: les métriques servant à orienter les cas de test, les métriques permettant de contrôler l'effort et les ressources et finalement les métriques servant à l'évaluation des performances de la stratégie basée sur le risque. Les

auteurs étudient ensuite dans [83], un cas empirique et montrent que les cas de test priorisés selon cette approche ciblaient les parties qui ont une plus forte probabilité de présence de fautes importantes. Les métriques de contrôle montrent en même temps que le processus n'est pas gourmand en temps, ce qui facilite son intégration dans le processus de test.

Jorgensen met en place en 2004 [83] un outil logiciel pour supporter les tests basés sur le risque. L'outil s'inspire de différents travaux de recherche [80,83,84] et se base sur l'analyse des cas d'utilisation, la détermination des facteurs (de risque), leur probabilité et leurs coûts. L'outil génère automatiquement les cas de test et les priorise selon le risque.

Chen et Probert [84] présentent, dans le cadre des tests de régression, une approche formelle de priorisation (de cas de test) basée sur l'analyse du risque et qui s'appuie sur les spécifications. Le risque y est évalué comme le produit de la probabilité de faute et du coût qu'elle engendre du point de vue du vendeur du logiciel (Ex. durée de la correction) et du point de vue de l'utilisateur (Ex. perte de marché). La démarche est exposée en quatre étapes: (1) Estimation des coûts associés à chaque cas d'utilisation, (2) Estimation de la sévérité et de la probabilité en fonction des défaillances et des sévérités passées, (3) Calcul du facteur d'exposition au risque pour chaque cas d'utilisation (à partir de 1 et 2), et finalement (4) Sélection/Tri des cas de test avec les plus grandes valeurs de facteurs d'exposition au risque.

Rosenberg et al. [85] ont imaginé une implémentation de la priorisation basée sur le risque pour les codes orientés objet au niveau des classes afin d'aider à l'orientation des tests logiciels OO à la NASA. L'idée est de se baser sur les métriques OO de CK dont le lien avec les fautes est avéré (démontré dans la littérature) pour estimer la probabilité de faute dans les classes logicielles et servir de premier facteur dans l'équation du risque. Le deuxième facteur (en l'occurrence la gravité) n'a pas fait l'objet d'analyse dans l'article, car nécessitant des connaissances approfondies du domaine d'application des logiciels analysés. Après analyse des dépôts de métriques des logiciels utilisés à la NASA, ils arrivent à la conclusion qu'une métrique seule ne devrait jamais être utilisée pour évaluer le risque.

La table d'une dizaine de classes publiées montre que le couplage et la complexité cyclomatique constituent de bons indicateurs de risque. Cette étude est la première à se baser directement sur le risque (estimé) des composants à tester pour prioriser les tests.

En effet, la priorisation suppose généralement que le testeur ait déjà choisi les composants ainsi que l'ensemble des scénarios. Cependant, selon Ray et Mohapatra [86], souvent peu de fautes sont à l'origine des défaillances dans les logiciels. De plus, ces fautes sont localisées dans une petite portion du code. L'idée de prioriser les composants à tester avant les cas de test permettrait alors d'augmenter les performances (le rendement) des techniques de priorisation des cas de test. Les auteurs s'attachent alors à prioriser les éléments du programme pour augmenter le rendement de l'effort global de test. L'objectif étant de localiser les parties critiques d'un code logiciel qui présentent de forts risques de fautes (à cause de leur complexité) et à impacts élevés (à cause de leur sévérité). L'impact en termes de sévérité est estimé par différentes métriques liées au domaine et au code source qui sont calculées par simulation contrôlée de fautes. Testée sur trois petits systèmes logiciels, cette approche augmente le rendement des tests et permet de cibler les fautes critiques du système.

Notre approche de priorisation basée sur le risque s'inscrit dans ce cadre et utilise l'analyse du risque pour sélectionner et prioriser les composants à tester. L'approximation du risque d'un composant est calculée uniquement à partir d'un ensemble de métriques obtenues suite à une analyse statique du code source. L'ensemble est formé essentiellement des métriques CK et d'une métrique synthétique capturant à la fois plusieurs attributs logiciels. Enfin, nous avons effectué la validation empirique sur des systèmes *open source* de grande taille et avons exploré la possibilité d'utiliser les modèles issus de projets différents pour l'orientation et la priorisation.

CHAPITRE 3. COLLECTE DE DONNÉES ET MÉTHODES D'ANALYSE

Pour les besoins de nos investigations, nous avons utilisé différents systèmes logiciels, différentes métriques de code source, différentes méthodes d'analyse statistique et d'apprentissage automatique grâce à des logiciels (outils) de calcul ainsi que des librairies et des environnements de développement que nous présentons dans ce chapitre. Le but étant essentiellement de présenter, de façon intégrée, les éléments de base des investigations qui seront traitées dans les chapitres suivants.

3.1 Les systèmes logiciels

Pour les besoins de notre étude (différents niveaux), nous avons extrait des données de plusieurs systèmes orientés objet, *open source* écrits en JAVA. Ces systèmes logiciels ont été développés par des équipes différentes et leurs domaines d'applications sont tout aussi différents. Une partie des classes de ces logiciels a été testée (écriture explicite de tests) unitairement en utilisant le Framework JUnit [59]. Nous les présentons ici par ordre croissant du nombre de classes.

- IO: La librairie Commons IO [87] est développée par Apache Software Foundation (ASF). Il s'agit d'une bibliothèque de fonctionnalités utilitaires standards qui permet de manipuler les entrées/sorties des différents systèmes d'exploitation. La version 2.1 utilisée dans nos analyses renferme près de 7 600 lignes de code contenues dans 104 classes logicielles.
- MATH: La librairie Commons Math [88] est développée elle aussi par l'ASF. Elle est très légère et contient des fonctionnalités mathématiques et statistiques les plus communes qui ne sont pas disponibles dans la bibliothèque Math de base du langage JAVA. Tous les algorithmes sont documentés et suivent les meilleures pratiques généralement acceptées. Cette bibliothèque n'a pas de dépendances externes au-delà du langage JAVA. La version 1.1 considérée dans nos expérimentations contient près de 8100 lignes de code réparties dans plus de 125 classes logicielles.
- JODA: La librairie Joda-time [89] regroupe un ensemble de fonctionnalités de manipulation de date écrites en JAVA. Elle est conçue comme une alternative complète aux

classes Date et Timestamp de JDK (Java Development Kit) et dispose d'une interface API intégrable pour divers systèmes de calendrier. Le déficit de fonctionnalités de manipulation de dates dans les versions de JAVA antérieures à la 8 a rendu Joda-time de facto le standard de manipulation de date pour ces anciennes versions de JDK. La version 2.0 utilisée ici contient près de 31 600 lignes de code réparties sur 225 classes logicielles.

- DBU: DbUnit [90] est un plug-in de JUnit destiné aux projets axés sur les bases de données. DBU restaure les bases de données dans des états connus entre les cycles de test. Ce qui permet d'éviter les multiples problèmes qui peuvent se produire lorsqu'un cas de test corrompt la base de données. DbUnit peut également aider à la vérification de la correspondance des données des bases de données avec les valeurs attendues. La version 2.1 sélectionnée contient 12 300 lignes de code réparties dans 238 classes logicielles.
- LOG4J: Log4j [91] est aussi développé par l'ASF. Il s'agit d'un framework souple et rapide permettant la journalisation des messages de débogage des logiciels. Il permet aussi d'activer la journalisation à l'exécution sans modifier le binaire d'application. Le paquetage log4j est conçu de sorte que ces déclarations peuvent rester dans le code livré sans encourir un coût pour la performance du système hôte. La version 1.2 sélectionnée contient près de 20 300 lignes de code réparties sur 252 classes logicielles.
- JFC: JFreeChart [92] est une bibliothèque JAVA graphique qui facilite, pour les développeurs, l'intégration de graphiques de qualité professionnelle dans leurs applications. Elle comprend une API (Application Programming Interface) cohérente et documentée, soutenant un large éventail de types de graphiques, une conception flexible qui est facile à étendre, et des applications qui ciblent à la fois le côté serveur et le côté client. JFreeChart supporte aussi différents types de sorties, y compris les composants Swing et JavaFX, et exporte les graphiques dans des formats variés d'images et de graphiques vectoriels. La version 1.0 sélectionnée contient près de 61 300 lignes de code contenues dans 496 classes logicielles.
- IVY: Apache Ivy [93] est une bibliothèque de gestion de dépendances intégrées à ANT, et orientée vers la gestion de dépendances JAVA. Elle peut, cependant, être utilisée

pour gérer les dépendances de toute sorte dans un projet logiciel. La version 2.2 sélectionnée contient près de 50 000 lignes de code réparties sur 610 classes logicielles.

- LUCENE: Apache Lucene [94] est une bibliothèque de recherche d'information textuelle complète de haute performance, entièrement écrite en JAVA. C'est une technologie appropriée pour presque toutes les applications qui nécessitent la recherche textuelle avancée, en particulier pour les systèmes multiplateformes. La version 2.2 sélectionnée contient près de 56 900 lignes de code réparties sur 659 classes logicielles.
- ANT: Apache Ant [95] est une bibliothèque et une ligne de commande utilitaire écrite en JAVA dont le rôle est de piloter le lancement de processus décrits dans des fichiers XML comme des objectifs et des points d'extensions interdépendants. La principale utilisation connue d'Ant est la construction d'applications JAVA à la manière des *makefiles* pour le langage C. Ant fournit un certain nombre de tâches intégrées permettant de compiler, assembler, tester et exécuter des applications JAVA. La librairie peut également être utilisée pour construire des applications non-JAVA, et plus généralement, pour piloter tout type de processus qui peut être décrit en termes d'objectifs et de tâches. Nous avons sélectionné 5 versions majeures d'ANT de 1.3 à 1.7 et une version mineure 1.5.3. *nightly built* (branche spéciale), contenant en plus du noyau, des définitions de tâches supplémentaires. Cette dernière est référencée par ANT (sans indication de version) dans le document. Les 6 versions confondues d'ANT cumulent près de 680 000 lignes de code réparties dans 2408 classes.
- POI: Apache POI [96] est une API JAVA permettant de manipuler différents formats de fichiers basés sur les normes Office Open XML (OOXML) et OLE 2, format qu'utilisent les documents de la suite office de Microsoft (Excel, Word et PowerPoint). La version 3.0 sélectionnée contient près de 136 000 lignes de code réparties sur 1540 classes logicielles.

Nous avons un total de 15 systèmes (9 systèmes différents et 6 versions différentes d'ANT) contenant plus d'un 1.2 million de lignes de code réparties sur 6160 classes. De ces systèmes, nous détenons le code source des classes logicielles et des suites de tests unitaires JUnit écrits par les différentes équipes de développement.

3.2 Les métriques

Pour évaluer les attributs logiciels, nous avons eu recours aux métriques logicielles proposées (et largement utilisées) dans la littérature. Ces métriques capturent différentes caractéristiques des classes telles que le couplage, la cohésion, la complexité, l'héritage et la taille. Certaines de ces métriques ont été utilisées pour évaluer des classes logicielles, nous les désignerons par métriques logicielles. D'autres ont servi à évaluer les classes de tests unitaires JUnit, nous les désignerons par métriques de test ou suite de métriques de test. Nous exposons, dans cette section, leurs descriptions.

3.2.1 Métriques logicielles

Les métriques logicielles orientées objet mesurent les attributs des classes logicielles. La plupart de ces métriques (noyau) sont issues de la suite de métriques OO de Chidamber et Kemerer (CK) [37]. Par ailleurs, certaines métriques présentées ci-dessous ont été adaptées au contexte orienté objet par différents auteurs, car étant initialement définies dans le cadre du paradigme procédural.

3.2.1.1 Métriques de couplage

- FANIN: Cette métrique compte le nombre de classes qui référencent directement la classe mesurée. Elle a été définie la première fois par Henry et al. [97] dans un cadre procédural, comme le nombre de flux se terminant dans un module plus le nombre de structures de données desquelles le module récupère de l'information. La métrique FANIN est surtout utilisée pour indiquer les modules critiques qui nécessitent une restructuration ou un effort de test particulier [85]. Pour une classe logicielle, le FANIN est aussi appelé couplage entrant et reflète son niveau d'utilisation par les autres classes du système. Un FANIN trop élevé est souvent signe d'une forte dépendance du système vis-à-vis de la classe logicielle.
- FANOUT: Cette métrique compte pour une classe le nombre d'autres classes qu'elle référence directement. Comme le FANIN, elle a été définie la première fois par Henry et al. [97], dans un cadre procédural. Elle comptait alors le nombre de flux issus d'un module plus le nombre de structures de données que le module met à jour. Appelée aussi couplage

sortant en orienté objet, cette métrique reflète le niveau de dépendance de la classe mesurée vis-à-vis des autres classes du système. Une dépendance trop élevée affecte la testabilité et la maintenance de la classe.

- CBO: Cette métrique appartient à la suite CK [37] et détermine, pour une classe logicielle donnée, le nombre de classes auxquelles elle est couplée et vice versa. Il s'agit d'une sommation des métriques FANIN et FANOUT. Elle évalue théoriquement l'interdépendance entre les classes du système. Briand et al. [99] ont montré qu'un couplage excessif entre les classes nuit à la modularité et constitue un obstacle à la réutilisation. Plus une classe est indépendante, plus il est facile de la réutiliser dans une autre application. Afin d'améliorer la modularité et de favoriser l'encapsulation, le couplage des classes doit être le plus limité possible. Plus le nombre de couples est élevé, plus les différentes parties du système sont sensibles aux modifications et plus la maintenance est difficile. Par ailleurs, la mesure du couplage s'avère utile pour prévoir le niveau de complexité des tests des différentes parties du système.

3.2.1.2 Métriques de cohésion

- LCOM: La métrique LCOM est issue de la suite de CK [37] et évalue théoriquement le manque de cohésion dans une classe logicielle. LCOM donne la différence entre le nombre de paires de méthodes ne partageant pas de variables d'instances et le nombre de paires de méthodes qui partagent des variables d'instances dans une classe logicielle. Si cette différence est négative, LCOM est fixée à zéro. LCOM évalue ainsi le manque de cohésion d'une classe. Une classe est cohésive si ses méthodes agissent sur le même ensemble de données. Les méthodes sont donc reliées entre elles. LCOM est censée renseigner sur la qualité de la structure de la classe. Une faible cohésion indique éventuellement que la classe a été investie de responsabilités disparates. La cohésion évalue aussi l'encapsulation des données et des fonctionnalités dans une classe et son niveau de réutilisabilité. Sa faiblesse est souvent un signe de défaut de conception. Une classe peu cohésive devant sans doute être éclatée en plusieurs classes plus cohésives (remaniement).

3.2.1.3 Métriques de taille

- LOC: Cette métrique compte le nombre de lignes d'instructions dans la classe mesurée. LOC évalue la taille d'une classe. La taille est fortement liée à la complexité. Une classe de trop grande taille est souvent synonyme de responsabilités disparates et de forte probabilité de présence de fautes, ce qui suggère que ces classes doivent être restructurées.

3.2.1.4 Métriques de complexité

- WMC: Cette métrique est issue de la suite CK. Elle somme les complexités cyclomatiques de toutes les méthodes de la classe mesurée. La complexité cyclomatique d'une méthode est donnée par la formulation de McCabe [100] comme étant le nombre de chemins linéairement indépendants qu'elle contient. WMC reflète donc la complexité globale d'une classe. Elle est liée à l'effort nécessaire au développement et à la maintenance d'une classe logicielle. Une valeur élevée est synonyme d'un risque élevé de présence de fautes dans la classe, mais aussi d'une faible compréhensibilité du code [36]. En tenant compte du nombre de méthodes, la complexité cyclomatique est aussi liée à une autre perspective de la taille des classes logicielles.

- RFC: Issue de la suite de CK, cette métrique compte, pour une classe donnée, le nombre de ses méthodes ainsi que le nombre de méthodes (des autres classes) appelées par la classe en réponse à un message. La métrique RFC reflète le niveau de communication potentiel entre une classe et les autres dont elle utilise les services. Cette métrique est théoriquement liée à la complexité et à la facilité de test d'une classe logicielle dans la mesure où plus une classe invoque des méthodes de diverses origines, plus ses fonctionnalités seront compliquées à vérifier. Une classe qui appelle un plus grand nombre de méthodes est considérée comme plus complexe et nécessitant plus d'effort de test.

3.2.1.5 Métriques d'héritage

- DIT: Cette métrique de la suite CK compte le nombre de classes qu'il y'a entre la classe mesurée et la racine de sa hiérarchie d'héritage. DIT indique la profondeur de l'arbre d'héritage. Elle évalue la complexité de la classe mesurée, mais aussi la complexité de la

conception de la classe. En effet, le comportement d'une classe étant plus difficile à comprendre quand le nombre de méthodes héritées augmente. Avec plus de classes héritées, la conception et la redéfinition des comportements deviennent plus délicates. Par ailleurs, cette métrique mesure aussi le niveau de réutilisation du code des classes dans la hiérarchie d'héritage. Plus une classe est loin (profonde) dans la hiérarchie, moins elle est générique.

- NOC: Cette métrique appartient à la suite CK. Elle compte le nombre de classes immédiatement dérivées de la classe mesurée. NOC reflète (théoriquement) l'impact potentiel d'une classe sur ses descendants. Elle évalue aussi le niveau de réutilisabilité de la classe mesurée dans la mesure où une classe ayant de nombreux enfants (classes dérivées) est très générique. Cependant, un très grand nombre de dérivations directes est signe de mauvaise conception et d'une abstraction impropre. Théoriquement, le nombre de dérivations directes impacte fortement la hiérarchie des classes et peut nécessiter un effort de test particulier pour la classe mesurée.

Notons que toutes les métriques logicielles utilisées ici ont été calculées grâce au plugin QA-metrics intégré dans l'environnement de développement Borland Together 2007 [88].

3.2.2 Métriques de test

Les métriques de test évaluent certains attributs des classes tests JUnit qui sont développées pour tester unitairement des classes logicielles. Elles nous permettront d'évaluer, entre autres, l'effort d'écriture et de conception des tests unitaires.

- TLOC: Cette métrique compte le nombre de lignes de code d'une classe test JUnit. Elle a été introduite par Bruntink et Van Deursen [101-102] pour évaluer la taille des classes tests unitaires JUnit.

- TASSERT: Cette métrique compte le nombre d'assertions (invocations des méthodes *assert()*) JUnit dans la classe test. Les assertions sont, en fait, utilisées par les testeurs pour comparer le comportement attendu de la classe sous test à son comportement effectif. Cette métrique, introduite aussi par Bruntink et Van Deursen. [101,102], est utilisée pour

indiquer une autre perspective de la taille d'une suite de test. Elle est directement liée à la construction du cas de test. Bruntink et Van Deursen ont basé la définition des métriques TLOC et TASSERT sur les travaux de Binder [103].

- TNOO: Cette métrique compte le nombre de méthodes dans une classe test [104]. Elle offre un autre point de vue de la taille de la classe de test JUnit.
- TDATA: Cette métrique compte le nombre de nouveaux objets JAVA créés dans une classe de test. Nous l'avons introduite dans [56] pour capturer la perspective de création des *mocks* qui sont des données factices nécessaires à l'initialisation des tests unitaires.
- TINVOK: Cette métrique compte le nombre d'invocations directes de méthodes dans une classe de test. Nous l'avons introduite dans [56] pour capturer la perspective de création de stubs (simulateurs) qui sont des classes factices simulant les comportements nécessaires au fonctionnement des classes logicielles sous test (une dimension reliée aussi aux tests d'intégration). TINVOK capture les dépendances nécessaires au fonctionnement de la classe pendant le test unitaire.
- THEFFEORT: Cette métrique fait partie de la famille de métriques de Halstead [105]. Elle a été utilisée pour mesurer l'effort nécessaire à la mise en œuvre d'une classe de test. Elle est proportionnelle au volume et au niveau de difficulté de la classe test. Nous supposons que cela reflète également la difficulté de la classe logicielle sous test et l'effort requis pour construire la classe de test unitaire correspondante.
- THDIFF: Cette métrique fait également partie de la famille de métriques de Halstead [105]. Elle a été utilisée pour indiquer le niveau de difficulté d'une classe de test. Elle est le produit du nombre d'opérateurs distincts par le nombre total d'opérandes rapporté au nombre d'opérandes distincts. Nous supposons que cela reflète aussi la difficulté de la classe sous test et l'effort global nécessaire pour construire la classe de test correspondante.

Notons que les métriques TLOC, TNOO, THEFFORT et THDIFF ont été calculées par le plugin QA-metrics intégré à l'environnement Borland Together 2007 [88] tandis que les métriques TASSERT, TINVOK et TDATA ont été calculées grâce à un plug-in de l'environnement de développement Eclipse [16] que nous avons développé. Le plug-in se

base sur la bibliothèque AST (Abstract Syntax Tree) de l'environnement Eclipse pour effectuer une analyse statique du code source des classes tests et compter le nombre d'invocations, d'assertions et de créations d'objets JAVA.

3.3 L'appariement des données

3.3.1 Données de test

Pour les besoins de nos investigations, nous avons eu à apparier les classes logicielles aux classes de tests unitaires qui leur ont été dédiées. La même approche a été utilisée dans des études empiriques antérieures qui ont abordé la problématique de l'évaluation de l'effort de test [90, 101, 102] unitaire (ou l'analyse des suites de tests).

JUnit [59] est un framework permettant l'écriture et l'exécution de tests unitaires automatisés pour des classes logicielles écrites en JAVA. Les cas de tests unitaires de JUnit sont écrits en JAVA par les testeurs. JUnit assiste les testeurs afin qu'ils puissent écrire les cas de test avec plus de commodité. Une utilisation typique de JUnit consiste à tester chaque classe logicielle du système à l'aide d'une classe test dédiée. Le test effectif de la classe logicielle est fait par sa classe test dédiée à travers l'appel de l'initialiseur de test JUnit. Après exécution, JUnit rapporte les tests qui ont réussi et les tests qui ont échoué.

Nous avons remarqué que les développeurs nommaient, généralement, les classes de test unitaires JUnit en ajoutant le préfixe ou le suffixe "Test" ou "TestCase" dans le nom des classes pour lesquelles des cas de test JUnit ont été développés. Seules les classes tests ayant de telles nomenclatures sont retenues dans nos analyses. Cette approche a déjà été adoptée dans d'autres études [107]. Nous avons alors observé, en analysant les classes tests JUnit des systèmes que dans certains cas, il n'y a pas une relation de 1 à 1 entre les classes tests JUnit et les classes logicielles testées. Cela a également été observé dans d'autres études antérieures [108,109]. Dans certains cas, en effet, plusieurs classes tests JUnit ont été reliées à une même classe logicielle testée.

Après avoir effectué le processus d'appariement selon cette méthode, nous avons fait refaire le processus à deux autres étudiants de maîtrise en informatique. Nous avons

comparé les résultats obtenus et avons remarqué quelques légères différences. Nous avons vérifié de nouveau les quelques résultats dans lesquels nous avons observé des différences pour finalement choisir les bons appariements en nous basant sur notre expérience et une analyse approfondie du code (source et test).

Nous utiliserons dans cette thèse le terme de *classe explicitement testée* pour désigner une classe logicielle pour laquelle une classe test unitaire JUnit dédiée a été écrite et pour laquelle, aussi, l'appariement a été effectué avec succès. Nous utiliserons aussi le terme d'*ensemble de test explicite* pour un système logiciel afin de désigner l'ensemble des classes explicitement testées.

3.3.2 Données de fautes logicielles

Pour le système ANT, nous avons recueilli les fautes trouvées dans 5 différentes versions du logiciel (1.3 à 1.7). Pour ce faire, nous avons fouillé les données issues du logiciel de suivi de fautes en ligne Bugzilla [110] dans lequel les bogues (erreurs dues aux fautes présentes dans le logiciel) d'ANT découverts durant les phases de test et d'utilisation sont consignés. Un bogue trouvé est signalé dans Bugzilla avec le statut NEW. Suivant sa description et son impact sur le système, un niveau de sévérité allant de trivial à bloquant (6 niveaux) lui est assigné. Il est ensuite analysé par la communauté des développeurs et trois situations peuvent se présenter: (1) Le bogue peut avoir déjà été signalé, auquel cas il prend le statut DUPLICATED. (2) Le bogue n'en est réellement pas un, auquel cas il est ignoré et ôté du système. (3) Il s'agit d'un vrai bogue reporté pour la première fois, auquel cas son statut passe de NEW à OPENED. Le bogue est par la suite attribué à la plus ancienne version d'ANT comportant l'erreur. Un développeur est alors assigné pour retrouver la (ou les) faute (s) et la (ou les) corriger. Le statut du bogue passe alors à l'état INPROGRESS. Le développeur assigné peut, au besoin, réévaluer le niveau de sévérité du bogue. Une fois le bogue corrigé, son statut change pour FIXED.

Pour retrouver la classe fautive, nous avons fouillé le fil des discussions qui suit la description du bogue (en général) ainsi que le patch apporté pour le corriger. Le patch contient les classes cibles des modifications qui permettent de corriger le bogue. L'analyse

du patch nous permet, en général, de déterminer la classe fautive. Dans certains cas, les bogues trouvés sont issus de l'interaction entre plusieurs classes. Nous assignons ces types de bogues aux différentes classes impliquées dans ces interactions. Dans d'autres cas (moins de 3% des cas), les bogues n'ont pas pu être assignés à des classes par manque d'informations ou de précisions sur leur origine. Ces bogues sont alors ignorés. Seuls les bogues ayant le statut FIXED ont été considérés dans nos analyses.

3.4 Les méthodes d'analyse

Nos démarches empiriques ont été supportées par différentes méthodes d'analyse statistique ou algorithmique des données que nous présentons dans cette section. Les analyses statistiques et l'apprentissage non supervisé ont été effectués avec le logiciel XLSTAT [111]. Il s'agit d'un plugin (extension) pour le logiciel EXCEL de Microsoft permettant d'effectuer des analyses statistiques avancées, directement sur les données des feuilles EXCEL. L'apprentissage supervisé et les validations croisées ont été effectués avec l'outil TANAGRA [112]. Tanagra est un logiciel gratuit implémentant plusieurs algorithmes d'apprentissage automatique.

3.4.1 Statistiques

3.4.1.1 Corrélations

Nous avons aussi utilisé deux mesures de corrélation pour analyser les liens entre certaines métriques.

- La corrélation de Pearson mesure l'intensité des liens entre deux variables. Elle quantifie la relation linéaire entre des variables aléatoires continues. Le calcul du coefficient de corrélation de Pearson repose sur le calcul de la covariance. Le coefficient de corrélation est, en fait, le rapport entre la covariance des deux variables aléatoires continues et le produit de leurs écarts-types. Il s'agit d'une standardisation qui permet d'obtenir une valeur entre -1 et +1, peu importe l'échelle des variables aléatoires. Une valeur proche de 1 (en absolu) est le signe que les deux variables aléatoires sont liées sans nécessairement une relation de cause à effet. Une valeur proche de 0 signifie que les deux

variables n'ont pas de liens significatifs. Nous calculerons en fonction des degrés de liberté une *valeur-p* pour chaque mesure de corrélation afin de vérifier que la probabilité que la valeur de la corrélation soit nulle par hasard est plus petite que 5%. Ce seuil de significativité (α = 5%) nous permettra de décider de la significativité ou non d'une valeur de corrélation.

- La corrélation de Spearman est un type de corrélation non paramétrique faisant appel aux rangs. Elle permet de mesurer le degré de relation linéaire entre deux variables à travers les rangs de leurs observations. Les rangs étant les numéros des positions des observations dans un tri croissant. Le coefficient de corrélation de Spearman aura une valeur comprise entre -1 et +1. Une corrélation positive est celle dans laquelle les rangs des observations des deux variables augmentent en même temps. Une corrélation négative est celle dans laquelle les rangs des observations de l'une des variables augmentent alors que les rangs des observations de l'autre diminuent. Une corrélation de +1 ou -1 indiquera que la relation entre les rangs est exactement linéaire alors qu'une valeur proche de zéro signifie qu'il n'y a pas de relation linéaire entre les rangs. Comme pour la corrélation de Pearson, nous calculerons pour chaque valeur de corrélation de Spearman une *valeur-p*, que nous comparerons au seuil typique α = 5% afin de déterminer sa significativité.

3.4.1.2 Régression linéaire

La régression linéaire est une technique statistique utilisée pour modéliser la relation entre une variable dépendante Y et des variables explicatives (indépendantes) X_i selon l'équation linéaire générale suivante:

$$Y = b_0 + b_1X_1 + b_2X_2 + \cdots + b_nX_n + \varepsilon$$

Dépendamment de la valeur de n, on parle de régression linéaire simple pour n = 1 ou de régression linéaire multiple pour n > 1 (plus d'une variable explicative). ε représente l'écart entre la valeur réelle de la variable dépendante et la valeur estimée par le modèle. Le coefficient b_0 est l'ordonnée à l'origine ou *intercept*. Le coefficient b_i donne le poids de la variable i dans l'explication de la variable Y. Pour chaque coefficient, nous calculerons la *valeur-p* que nous comparerons au seuil de significativité α = 5%. Pour évaluer la

significativité globale du modèle, nous calculerons le coefficient de détermination R^2 dont la valeur est comprise entre 0 et 1. Le R^2 s'interprète comme la proportion de la variabilité de la variable dépendante Y expliquée par le modèle. Plus le R^2 est proche de 1, meilleur est le modèle.

Nous n'utiliserons pas de régression linéaire pour faire de la prédiction, mais plutôt pour analyser le type de liens entre les variables dépendantes et indépendantes.

3.4.1.3 Régression logistique binaire

La régression logistique est une technique utilisée pour modéliser le lien entre des variables indépendantes et une variable dépendante binaire selon l'équation logistique suivante:

$$P(X_1, X_1, \dots X_n) = \frac{e^{(a+\sum_{i=1}^{n} b_i X_i)}}{1 + e^{(a+\sum_{i=1}^{n} b_i X_i)}}$$

Tout comme la régression linéaire, il y a deux types de régressions logistiques dépendamment du nombre de variables explicatives. On parle de régression logistique univariée en présence d'une seule variable explicative et de régression logistique multivariée dans le cas de plus d'une variable explicative.

P est la probabilité que la variable dépendante prenne la valeur de la modalité de référence. Les X_is sont les variables indépendantes. b_i est le coefficient de régression estimé (contribution approximative) correspondant à la variable indépendante X_i. Plus la valeur absolue du coefficient b_i est élevée, plus l'impact de la variable indépendante sur (la probabilité de) la variable dépendante est élevé. Nous considèrerons la version normalisée des coefficients b_is lors de nos différentes analyses. Nous calculerons, par ailleurs, pour chaque b_i la *valeur-p* qui représente la probabilité que le coefficient soit nul par hasard. Nous fixerons une valeur seuil typique α à 5% au-delà de laquelle le coefficient b_i sera considéré comme non significatif.

Pour évaluer les modèles logistiques générés, nous calculons le coefficient R^2 (Nagelkerke) qui est défini comme la proportion de la variance totale de la variable dépendante expliquée par le modèle. La valeur de R^2 est comprise en 0 et 1. Plus le

coefficient R^2 est élevé, plus l'effet des variables indépendantes sur le modèle est important. Le coefficient R^2 peut aussi être vu comme l'amélioration qu'apportent nos variables explicatives par rapport au modèle de base (ou modèle nul), sans variable explicative, modèle qui est uniquement basé sur la moyenne de la distribution des modalités (variable dépendante).

Pour évaluer la capacité de prédiction des modèles logistiques, il est possible d'utiliser une matrice de confusion. Il faut, cependant, fixer la probabilité seuil à partir de laquelle on décide qu'un évènement doit être considéré comme positif. Ce point de séparation (entre 0 et 1) permet de classifier une observation dans la modalité 1 ou 0, à partir de la probabilité calculée par le modèle obtenu. Pour éviter un choix arbitraire du point de séparation, nous préfèrerons analyser la courbe ROC (Receiver Operating Characteristics). On désigne par sensibilité la proportion d'évènements positifs bien classés, et par spécificité la proportion d'évènements négatifs bien classés. Si l'on fait varier le point de séparation entre 0 et 1, la sensibilité et la spécificité varient. La courbe des points (1-spécificité, sensibilité) est la courbe ROC.

L'AUC (Area Under Curve) représentant l'aire sous la courbe ROC, a une valeur entre 0 et 1. Une valeur proche de 1 signifie que le modèle s'ajuste parfaitement avec les données alors qu'une valeur proche de 0.5 indique qu'il ne performe pas mieux que le hasard. Quand l'AUC est supérieure ou égale à 0.70, on dit que le modèle est un bon prédicteur [113]. L'AUC permet de quantifier les performances d'un modèle, et de les comparer à celles d'autres modèles.

3.4.1.4 Régression logistique multinomiale

La régression logistique multinomiale est une généralisation de la régression logistique binaire. Elle est utilisée pour modéliser les observations dans lesquelles la variable dépendante contient plus d'une modalité non ordonnée ou polytomiques et les variables indépendantes (ou prédicteurs catégoriels) sont à valeurs réelles continues. La régression logistique multinomiale permet dans ce cas de comparer chaque catégorie de la variable de réponse à une catégorie de référence, en utilisant des modèles de régressions

logistiques binaires. Un modèle de régression logistique binaire compare une dichotomie, alors qu'un modèle de régression logistique multinomiale compare plusieurs dichotomies. Cette procédure produit alors, un certain nombre de modèles de régressions logistiques binaires qui comparent spécifiquement certaines catégories de réponse à une catégorie de référence. Pour un nombre *q* de catégories possibles dans la variable dépendante, le modèle sera constitué de *q-1* équations logistiques simultanées qui correspondent à:

$$log\left(\frac{P(cat_j)}{P(cat_r)}\right) = b_0^{(j)} + \sum_{i=1}^{k} b_i^{(j)} X_i \ , avec\ j = 1 \ldots . q-1$$

On peut voir dans l'expression ci-dessus que l'une des catégories (la *r*) est utilisée comme catégorie de base et est appelée *catégorie de référence*. Après avoir estimé les coefficients b_is du modèle par la méthode de maximum de vraisemblance, nous calculons les probabilités de chacune des catégories. La prédiction finale est la catégorie ayant la probabilité maximale. Pour évaluer la significativité du modèle, nous déterminons aussi le coefficient R^2 de Nagelkerk ainsi que la valeur-p (*-2log*). Finalement, nous calculerons pour chaque paramètre b_is le degré de significativité sous forme de *valeur-p*. Nous utiliserons la valeur typique de seuil α = 5%.

3.4.1.5 Analyse en composantes principales

L'Analyse en Composantes Principales (ACP) est l'une des méthodes d'analyse de données multivariées les plus utilisées dès lors que l'on dispose de données quantitatives (continues ou discrètes) dans lesquelles *n* observations sont décrites par *p* variables. Si *p* est assez élevé, il est difficile d'appréhender la structure des données et la proximité entre les observations avec les analyses de statistiques descriptives ou une matrice de corrélation. L'ACP peut être considérée comme une méthode de projection permettant de projeter les observations depuis l'espace à p dimensions (des *p* variables) vers un espace à *k* dimensions ($k < p$) tel qu'un maximum d'information soit conservé (l'information est ici mesurée en termes de variance totale des observations) sur les premières dimensions. Si l'information associée aux 2 ou 3 premiers axes représente un pourcentage suffisant de la variabilité totale des observations, on pourra représenter les observations dans un espace

de plus petite dimension (2 ou 3) pour en faciliter l'interprétation. L'ACP permet aussi de mettre en évidence le recoupement de l'information issue des *p* variables ainsi que les dimensions sous-jacentes qu'elles capturent.

Deux méthodes sont communément utilisées pour déterminer le nombre de facteurs à retenir pour l'interprétation des résultats: (1) Le Scree test [114] qui est fondé sur la courbe décroissante des valeurs propres. Le nombre de facteurs à retenir correspond au premier point d'inflexion détecté sur la courbe. (2) Le pourcentage cumulé de variabilité représenté par les axes factoriels. On peut alors décider de se contenter d'un certain pourcentage. Nous adopterons cette deuxième méthode pour limiter le nombre de facteurs principaux et fixerons le pourcentage cible à 80%.

La représentation des variables dans l'espace des *k* facteurs permet d'interpréter visuellement les corrélations entre les variables d'une part, et entre les variables et les facteurs d'autre part, moyennant certaines précautions. En effet, qu'il s'agisse de la représentation des observations ou des variables dans l'espace des facteurs, deux points très éloignés dans un espace à 3 dimensions (par exemple) peuvent apparaitre proches dans un espace à 2 dimensions en fonction de la direction de la projection (figure 1).

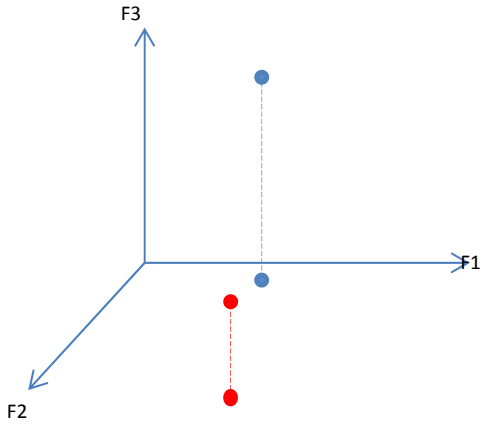


Figure 1: Effet de la projection dans des dimensions inférieures.

Pour éviter des interprétations erronées, on peut considérer que la projection d'un point sur un axe, un plan ou un espace à 3 dimensions est fiable si la somme des cosinus carrés sur les axes de représentation n'est pas trop éloignée de 1. Les cosinus carrés seront calculés dans certains cas pour éviter de mauvaises interprétations. Par ailleurs, nous

utiliserons le pourcentage de contribution des variables dans (ou leur corrélation avec) chaque nouvelle composante pour étudier la contribution relative des différentes variables qui ont servi à la construction des axes factoriels.

3.4.1.6 Test de la moyenne

Le test paramétrique Z permet de comparer les moyennes de deux échantillons de variances inconnues. La méthode de calcul est différente en fonction de la nature des échantillons. On distingue le cas où les échantillons sont indépendants, du cas où ils sont appariés. Le test Z est dit paramétrique, car il suppose que les échantillons sont distribués suivant des lois normales. Nous testerons cette hypothèse à l'aide d'un test de normalité. Pour comparer les moyennes de deux échantillons indépendants E_1, comprenant n_1 observations, de moyenne μ_1 et de variance s_1^2 et E_2 (indépendant de E_1), comprenant n_2 observations, de moyenne μ_2 et de variance s_2^2, nous supposons que la différence entre les moyennes est nulle. Le test détermine une *valeur-p* que l'on compare au seuil de significativité $\alpha = 5\ \%$. Elle est la probabilité que la différence δ entre les moyennes des deux échantillons soit égale à 0 par hasard. Nous calculerons le coefficient *z* pour le comparer à un seuil critique de référence pour une distribution normale.

3.4.2 Algorithmes d'apprentissage

3.4.2.1 Apprentissage non supervisé

Nous avons utilisé des algorithmes de clustering pour discrétiser certaines variables continues. Ces méthodes d'apprentissage automatique divisent des observations hétérogènes en catégories contenant des observations similaires de sorte que les observations de catégories différentes ne soient pas similaires. Il existe plusieurs approches permettant ce regroupement suivant les critères de similarité ciblés. Nous avons utilisé 2 approches pour nos expérimentations que nous exposons dans cette sous-section : Le KMEAN et le partitionnement univarié.

- KMEAN: Introduit par MacQueen [115], le clustering KMEAN présente plusieurs avantages. En effet, affectée à un cluster au départ du processus, une observation peut changer pour un autre cluster au cours des itérations, ce qui n'est pas le cas d'autres

techniques de clustering comme le regroupement hiérarchique ascendant. Par ailleurs, en augmentant les points de départ et les itérations, on peut explorer plusieurs solutions possibles. L'inconvénient de la méthode est qu'elle ne permet pas de découvrir automatiquement un nombre optimal de clusters cohérents pour un jeu de données. Il faudra lui assigner un nombre de clusters à l'initialisation. Le regroupement KMEAN est en fait un problème d'optimisation qui converge, quel que soit le point de départ, vers une solution. Pour la première itération, un point de départ est choisi. Cela consiste à associer le centre des k clusters à k objets pris au hasard. On calcule ensuite la distance euclidienne entre les objets et les k centres puis on affecte les objets aux centres dont ils sont les plus proches. Ensuite, on redéfinit les centres à partir des objets qui ont été affectés aux différents clusters. Puis, on réaffecte les objets en fonction de leur distance aux nouveaux centres, et ainsi de suite, jusqu'à ce que la convergence soit atteinte. Il existe différents critères d'optimisation. Nous avons choisi le déterminant W de la matrice de covariance intra-classe commune. Minimiser ce déterminant pour un nombre de cluster donné, revient à minimiser la variance intra-classe, donc l'hétérogénéité des groupes. Ce critère est moins sensible aux effets d'échelle des variables, mais peut rendre la taille des groupes moins homogène qu'avec d'autres critères.

- Partitionnement univarié: Cette technique de partitionnement consiste à regrouper N observations décrites par une seule variable quantitative en k clusters homogènes. L'homogénéité est mesurée en termes de somme des variances intra-classe. Pour maximiser l'homogénéité des clusters, on doit minimiser la somme des variances intra-cluster. L'algorithme utilisé par notre outil de calcul s'appuie sur la méthode proposée par W.D. Fisher [116]. Le partitionnement univarié peut être vu comme une discrétisation d'une variable quantitative en une variable ordinale.

3.4.2.2 Apprentissage supervisé

Nous avons utilisé des algorithmes de classification supervisée pour construire des modèles de prédiction. Ces algorithmes partent des observations et de leurs classifications existantes comme données d'entrainement pour extrapoler et classifier de nouvelles

observations (qui leur sont jusque-là inconnues). Différentes techniques de validation croisées permettent d'évaluer leurs performances.

- Algorithme C4.5: Cet algorithme génère un arbre de décision à partir d'un ensemble de données d'entrainement en utilisant le critère d'entropie. À chaque nœud de l'arbre, l'algorithme choisit l'attribut des données qui divise le plus efficacement son ensemble d'échantillons enrichis en sous-ensembles dans une classe ou dans l'autre. Le critère de division est le gain d'information normalisé (différence d'entropie). L'attribut ayant le gain d'information normalisé le plus élevé est choisi pour prendre la décision. L'algorithme C4.5 se poursuit récursivement pour les nœuds enfants. Cet algorithme a été proposé par Quinlan [117].
- KNN: L'algorithme des K plus proches voisins est une méthode de classification et de régression non paramétrique permettant de reconnaitre des patrons dans les observations. Il consiste à estimer la classe d'une nouvelle entrée E en prenant en compte, et avec le même poids, les k échantillons d'entrainement les plus "proches" de la nouvelle entrée E. La proximité est déterminée par une distance. Dans nos expérimentations, nous utiliserons la distance HEOM (Heterogeneous Euclidean-Overlap Metric). Nous choisirons différentes valeurs de K pour tester différentes configurations de l'algorithme.
- BNAIF: Le Bayes naïf est un type de classification bayésienne probabiliste basé sur le théorème de Bayes [24] supposant une forte hypothèse sur les descripteurs (conditionnellement indépendants) d'où le qualificatif naïf. Un avantage de cette méthode est la simplicité de programmation, la facilité d'estimation des paramètres et sa rapidité même sur de très grandes quantités de données. Sur un ensemble d'apprentissage, l'approche consiste à: (1) déterminer les probabilités a priori de chaque classe (analyse de fréquence), (2) à appliquer la règle de Bayes pour déterminer les probabilités a posteriori des classes, et (3) à choisir la classe la plus probable.

3.4.2.3 Ensemble de classificateurs supervisés

Nous avons exploré les techniques d'amélioration des performances des classificateurs. Nous en avons utilisé deux que nous présentons dans cette sous-section.

- Le BAGGING: Cette méthode de dopage [24] consiste à tirer uniformément et avec remise les observations afin de créer k nouveaux échantillons de même taille. L'apprentissage sur ces nouveaux échantillons permet d'obtenir k modèles du classificateur. On obtient le modèle final par vote de la majorité des k modèles intermédiaires.

- Le BOOSTING: Sur un échantillon d'apprentissage *S* de taille *m*, cette technique de dopage [118] consiste à utiliser un classificateur sur trois sous-ensembles d'apprentissage. On obtient d'abord une première hypothèse h_1 sur un sous-échantillon S_1 d'apprentissage de taille $m_1 < m$. On apprend alors une deuxième hypothèse h_2 sur un échantillon S_2 de taille m_2 choisi dans le sous-ensemble $S - S_1$ dont la moitié des exemples est mal classée par h_1. On apprend finalement une troisième hypothèse h_3 sur m_3 exemples tirés dans $S - (S_1 \cup S_2)$ pour lesquels h_1 et h_2 sont en désaccord. L'hypothèse finale est obtenue par un vote majoritaire des trois hypothèses apprises.

3.4.2.4 Validation des algorithmes d'apprentissage supervisé

- Le fitting: Cette validation consiste à tester les algorithmes sur les données d'entrainement. Les résultats semblent en général meilleurs (les erreurs plus petites) que pour les autres approches de validation. Cela s'explique simplement par le fait que les données de test ont servi à créer le modèle. Cette validation n'est pas robuste, mais permet de voir à quel point le modèle produit s'ajuste aux données d'apprentissage. Un fort lien (faible erreur) peut être un indice de mauvaise prédiction sur de futures données de test. on parle de surapprentissage (overfitting). Le surapprentissage nuit grandement à la capacité de prédiction générale du modèle sur d'autres données de test.

- 10-FV: Le 10-Fold-Validation est une technique de validation beaucoup plus robuste que la précédente. Elle consiste à partitionner l'échantillon global en 10 sous-ensembles disjoints dont 9 serviront en même temps d'ensemble d'entrainement et le $10^{ème}$ d'ensemble de test. Le processus est reproduit 9 autres fois avec, à chaque itération, le choix d'un ensemble-test différent parmi les ensembles du partitionnement initial. Cette validation est un cas particulier du K-Fold Validation, avec une partition initiale de

l'échantillon en K sous-ensembles. Au bout du traitement, une erreur moyenne est compilée.

- LOOV ou Leave-One-Out Validation est un cas particulier du K-Fold-Validation où K = N-1 et N est la taille de l'échantillon global. À chaque itération, une instance (observation) servira de test et les N-1 instances serviront d'ensemble d'entrainement. Cette méthode est couteuse en calcul, mais fournit une bonne approximation de l'erreur globale.
- 10x70/30: Est une technique de validation avec une division aléatoire du jeu de données initial en ensemble d'apprentissage de 70% et ensemble de test 30%. Le processus est répété 10 fois de manière aléatoire. Il s'agit d'une méthode de validation applicable avant et après les techniques de dopage.

CHAPITRE 4. INDICATEURS DE QUALITÉ

4.1 Introduction

Dans cette section, nous faisons un bref rappel sur les indicateurs de qualité (Qi), que nous avons introduits en [119]. La métrique Qi est essentiellement basée sur le concept de graphe de contrôle réduit aux appels (Control Call Graph -CCG), qui est une forme réduite du concept connu sous le nom de graphe de flux de contrôle (Control Flow Graph - CFG). Le CCG est, en fait, un CFG dans lequel les nœuds qui représentent les instructions (ou blocs d'instructions séquentielles) ne contenant pas d'appels de méthodes sont retirés. La métrique Qi est normalisée et ses valeurs sont dans l'intervalle [0, 1]. Une valeur faible de Qi indique qu'une classe est à haut risque et a besoin d'un effort de test relativement plus soutenu pour assurer sa qualité. Une forte valeur de Qi, c.-à-d. proche de 1, indique que cette classe est à faible risque. Cela pourrait signifier que la classe est relativement peu complexe, ou complexe avec un effort de test relativement important et/ou proportionnel à sa complexité.

4.2 Graphe de contrôle réduit aux appels (CCG)

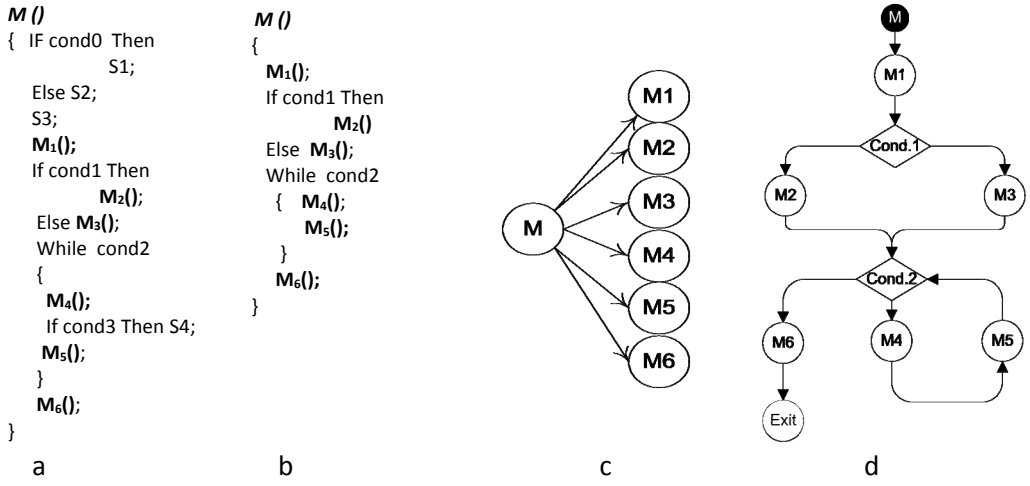


Figure 2: Du code au CCG.

Dans l'exemple de la méthode M représentée dans la figure 2.a, les *Si* représentent des blocs d'instructions qui ne contiennent pas d'appels de méthodes. Le code de la méthode M réduit aux appels est donné par la figure 2.b. Les instructions ou blocs d'instructions ne contenant pas d'appels de méthodes ont été retirés du code de la méthode M. La figure 2.c donne le graphe d'appels correspondant, qui ignore les chemins du flux de contrôle dans la méthode. La figure 2.d qui représente le graphe de contrôle réduit aux appels correspondant, qui contrairement au graphe d'appels simple, tient compte des chemins du flux de contrôle dans la méthode. Contrairement au graphe d'appels traditionnel, le CCG est un modèle beaucoup plus précis. Les CCG capturent la structure des appels et les contrôles qui leur sont reliés.

4.3 Polymorphisme et graphe de contrôle réduit aux appels

Le polymorphisme est un concept important de la programmation orientée objet. Il est aussi sans doute le concept le plus difficile à capturer par une simple analyse statique du code source. En effet, l'appel effectif vers les méthodes virtuelles n'est connu qu'à l'exécution du code à cause des liaisons retardées (dynamiques). Pour tenir compte du polymorphisme dans le CCG, nous avons choisi de représenter les appels polymorphes par des polygones de n+1 cotés, *n* étant le nombre de méthodes virtuelles pouvant éventuellement correspondre à cet appel. La liste des méthodes capables de répondre à un appel polymorphe est obtenue par analyse statique du code source. En supposant par, exemple, dans la figure 1.a que l'appel de M2 est polymorphique et que M21 correspond aussi à cet appel, nous obtenons le CCG suivant:

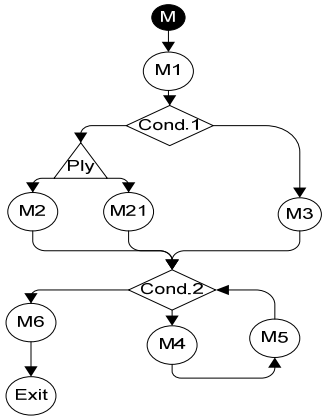


Figure 3: Représentation du polymorphisme dans le CCG.

4.4 Formalisme des indicateurs d'assurance qualité

Dans [119], nous avons défini l'indicateur d'assurance qualité (Qi) d'une méthode M comme une sorte d'estimation de la probabilité que le flux de contrôle traverse le code de la méthode sans produire d'erreur. Il peut être considéré comme un indicateur de risque associé à une méthode (et à un plus haut niveau, à la classe logicielle). Le Qi d'une méthode Mi dépend des caractéristiques intrinsèques de la méthode, comme sa complexité cyclomatique, son taux de couverture de test (l'effort de test réel investi sur la méthode Mi), et des Qi des autres méthodes que Mi appelle. Nous supposons, en fait, que la qualité d'une méthode en termes de fiabilité dépend aussi de la qualité des autres méthodes avec lesquelles elle collabore (dépend) pour réaliser sa tâche. En effet, dans les systèmes logiciels orientés objet, les objets collaborent pour réaliser leurs responsabilités respectives à travers des appels de méthodes (interactions). Dans ce cas, une méthode de faible qualité peut impacter directement ou indirectement de manière négative la méthode qui utilise ses services. Il s'agit, alors, d'une sorte de propagation dépendamment de la distribution du flux de contrôle à travers le système. Il n'est pas évident, particulièrement dans le cas de systèmes d'envergure, d'identifier intuitivement ce type d'inférences entre les classes. Ce type d'information n'est pas directement capturé par les métriques de couplage OO. En effet, elles ne capturent que les liens directs entre les classes des systèmes.

Nous avons formulé les Qi pour tenir compte de cette propagation dépendamment du chemin du flux de contrôle à travers les objets. Le Qi d'une méthode M_i est donné par la formule suivante:

$$Qi_{M_i} = Qi^*_{M_i} * \sum_{j=1}^{n_i} \left[P(C_j^i) \prod_{M \in \sigma_j} Qi_M \right] \quad (1)$$

Avec

Qi_{M_i}: Indicateur de qualité de la méthode M_i,

$Qi^*_{M_i}$: Indicateur de qualité intrinsèque de la méthode M_i (c.f. 4.5),

$P(C_j^i)$: Probabilité de passage par le chemin C_j^i de la méthode M_i,

Qi_M: Indicateur de qualité d'une méthode M du chemin C_j^i, dans le CCG de M_i,

σ_j: Ensemble de méthodes invoquées dans le chemin C_j^i,

et enfin n_i: Nombre de chemins possibles dans la méthode M_i.

L'application de cette formule (1) aux N méthodes d'un système logiciel génère un système de N équations à N inconnues. Il s'agit d'un système non linéaire d'équations polynomiales à plusieurs variables.

L'exemple suivant nous donne une idée de la forme des équations obtenues pour la méthode M de la figure 1.

$$Qi_M = Qi^*_M * \left[Qi_{M_1} * \left(0.5Qi_{M_2} + 0.5Qi_{M_3} \right) * \left(0.75Qi_{M_4} * Qi_{M_5} * Qi_{M_6} + 0.25Qi_{M_6} \right) \right]$$

5.1.1. Règles d'affectation des probabilités

Le graphe de contrôle réduit aux appels d'une méthode peut être vu comme un ensemble de chemins que le flux de contrôle peut emprunter. Le passage par un chemin particulier dépend à l'exécution de l'état des conditions à l'intérieur des structures de

contrôle. Pour tenir compte de cet aspect probabiliste du flux de contrôle, nous avons assigné des probabilités à chaque chemin du graphe de contrôle comme suite:

$$P(C) = \prod_{A \in \theta} P(A) \quad (3)$$

θ étant l'ensemble des arcs directs (sans structures de contrôle) composant le chemin C est $P(A)$ la probabilité que l'arc A soit traversé par le flux de contrôle à la sortie d'une structure de contrôle.

Pour faciliter les calculs, nous avons assigné des probabilités fixes aux différentes structures de contrôle du langage JAVA selon les règles de la table 1. Cependant, ces règles peuvent être assignées par un programmeur connaissant bien le code ou encore par une analyse dynamique du code.

Table 1: Règles d'affectation des probabilités aux structures de contrôle.

Structure de contrôle	Règles d'affectation
(if, else)	0.5 sortie de l'arc avec "condition = true"
	0.5 sortie de l'arc avec "condition = false"
while	0.75 sortie de l'arc avec "condition = true"
	0.25 sortie de l'arc avec "condition = false"
(do, while)	1 pour l'arc: "do" est exécuté au moins 1 fois
(switch, case)	1/n pour chaque arc des n "cases"
(?,:)	0.5 sortie de l'arc avec "condition = true"
	0.5 sortie de l'arc avec "condition = false"
for	0.75 entrée de la boucle
	0.25 saut de la boucle
(try, catch)	0.75 pour l'arc formé du bloc "try"
	0.25 pour l'arc formé du bloc "catch"
Polymorphisme	1/n pour chaque appel de méthode éventuel

4.5 Indicateurs de Qualité intrinsèques

L'indicateur de qualité intrinsèque d'une méthode regroupe certaines caractéristiques internes à la méthode. Parmi les éléments caractéristiques du Qi intrinsèque figurent: la complexité cyclomatique et l'effort de test unitaire (taux de couverture de test). L'indicateur de qualité intrinsèque pour une méthode est formellement défini comme suit:

$$Qi^*_{M_i} = (1 - F_i) \quad avec \quad F_i = \frac{(1 - tc_i) * CC_i}{CC_{max}} \quad et \quad CC_{max} = \max_{1 \le i \le N}(\mathrm{CC}_i)$$

Avec tc_i comme taux (en pourcentage) de couverture de test unitaire et CC_i, la complexité cyclomatique de la méthode M_i.

Les études [39,42,49,50] montrent l'existence de liens significatifs entre la complexité cyclomatique et la probabilité de présence de fautes dans les classes logicielles. L'activité de test réduit ce risque lié à la complexité et contribue à améliorer la qualité générale du composant.

4.6 Calcul du Qi

En reprenant l'équation (1) pour les N méthodes d'un système logiciel, nous obtenons un système de N équations non linéaires à N inconnues. Ce système peut être vu comme un problème de point fixe pour une fonction F de $\mathbb{R}^N \to \mathbb{R}^N$ définie comme suite:

$$\mathbb{R}^N \to \mathbb{R}^N$$

$$X \to F(X) = \big(f_1(X), f_2(X), f_3(X), \dots, f_N(X)\big)$$

$$f_i(X) = Qi^*_{M_i} * \sum_{j=1}^{n_i} \left[P\big(C_j^i\big) \prod_{k \in \sigma_j} x_k \right]$$

Alors, l'équation $F(X) = X$ est un problème de point fixe. L'existence et l'unicité de la solution nous est garantie par le théorème du point fixe de Banach, car F est une fonction lipchitzienne, c'est à dire contractante de rapport $K < 1$. En posant $X_n = F(X_{n-1})$, la suite $(X_n)_n$ converge alors vers la solution. La méthode des approximations successives nous permet d'approcher la solution. Si on note par X^* la solution exacte, l'erreur à la nième itération est donnée par

$$e_n = \|X_n - X^*\| = \|F(X_{n-1}) - F(X^*)\| \le K\|X_{n-1} - X^*\| \dots \le K^n\|X_0 - X^*\| \le K^n$$

K étant la constante de Lipschitz.

Pour avoir une erreur $\le 10^{-5}$ il faudra $n \ge -5 * \frac{\ln(10)}{\ln(K)}$ itérations.

Après résolution du problème, nous calculons les indicateurs de qualité des classes. Le Qi_C d'une classe C est défini comme le produit des Qi de ses méthodes publiques:

$$Qi_C = \prod_{M \in \delta} Qi_M \quad (2)$$

δ étant l'ensemble des méthodes publiques de la classe C. Le calcul des Qi est entièrement automatisé par un plug-in d'Eclipse que nous avons implémenté pour le langage JAVA.

CHAPITRE 5. INDICATEURS DE QUALITÉ ET EFFORT DE TEST

5.1 Introduction et Objectifs

Un grand nombre de métriques logicielles a été proposé dans la littérature [37,105,120]. Elles sont utilisées pour estimer différents attributs logiciels tels que la taille, la complexité, le couplage, la cohésion, etc. Il n'est, cependant, pas évident pour un responsable de projet ou un développeur de choisir les métriques adéquates face à une problématique donnée dans un projet logiciel. Par ailleurs, les informations capturées par les métriques orientées objet classiques se recoupent, ce qui rend difficiles leur analyse et leur interprétation [42,121], sans compter les problèmes liés à leur gestion. Le besoin de développer des métriques synthétiques est devenu de plus en plus évident dans la littérature. Les Qi capturent de façon synthétique plusieurs attributs tels que la taille, la complexité et le couplage. Ils fournissent une valeur numérique normalisée permettant aux responsables de projet (ou développeurs) d'avoir un aperçu global de la criticité des composants du système sous analyse.

Pour déterminer à quel point les indicateurs de qualité capturent l'information fournie pas les autres métriques OO traditionnelles, nous avons effectué dans [119] une étude d'envergure comparant les Qi à 9 métriques de design orientées objet issues de la suite CK ainsi que la métrique classique de taille (nombre de lignes de code). Pour ce faire, nous avons effectué une analyse par composantes principales des informations de 7 systèmes logiciels open source, capturées par 10 métriques et les résultats ont montré que les Qi couvraient entre 69.79 et 82.11 % de l'information fournie par les métriques de couplage, d'héritage, de complexité cyclomatique, de cohésion et de taille. Ces résultats confirment le premier objectif des indicateurs de qualité à savoir: capturer l'information fournie par certaines métriques orientées objet de la suite CK.

L'autre objectif des Indicateurs de qualité est d'offrir une vision globale très synthétique du logiciel en mettant en évidence les classes les plus critiques dans un système. En effet, la métrique Qi détermine pour chaque classe logicielle, une grandeur relative utilisable pour l'identification des classes les plus critiques, nécessitant un effort de

test relativement plus élevé. Cette mesure permet aussi de comparer les classes les unes par rapport aux autres. Afin de déterminer le lien effectif entre les Qi et l'effort de test unitaire des classes, nous avons mené plusieurs travaux d'envergure qui ont déjà fait l'objet de publications [57,122,123].

Dans ce chapitre, nous explorons en général, le lien entre les métriques logicielles et l'effort de test et en particulier le lien entre les indicateurs de qualité et l'effort de test. Les différentes investigations confirment que le Qi donne de bons résultats, dans le pire des cas comparables à ceux donnés par la métrique de taille classique LOC.

Notons que parallèlement à ces recherches, les indicateurs de qualité ont été utilisés dans [124,125] pour analyser l'évolution de la qualité des logiciels. Les résultats montrent là aussi que les Qi se dégradent et s'améliorent avec la (dégradation et l'amélioration de la) qualité du logiciel.

5.2 Évaluation de l'effort de test unitaire

La facilité de tester un logiciel ou un composant logiciel est connue en génie logiciel sous le concept général de la testabilité. Il s'agit d'une notion complexe qui ne découle pas d'une propriété intrinsèque d'un artéfact logiciel et ne peut, donc, pas être mesurée directement comme des attributs simples tels que la taille, la complexité cyclomatique ou le couplage. En effet, selon certains auteurs [107,126-128], en plus des caractéristiques du logiciel, la testabilité est influencée par des facteurs humains, les environnements, la technique et les processus utilisés ce qui complique davantage son évaluation.

La testabilité a été définie par IEEE [129] comme le degré de facilité (pour un système ou un composant) avec lequel on peut établir des critères de test, et réaliser des tests permettant de vérifier la satisfaction de ces critères.

Fenton et Pfleeger [130] ont défini la testabilité comme un attribut externe du logiciel. Freedman [131] définit quant à lui deux composantes pour mesurer la testabilité: l'observabilité (caractérisant la facilité d'observer un écart entre le comportement du logiciel et les spécifications lors des tests) et la contrôlabilité (caractérisant la facilité de

générer des entrées qui causent ces écarts de comportement). Voas [132] voit la testabilité comme une probabilité pour un cas de test d'échouer en présence de fautes dans le logiciel. Voas et Miller [133] proposent une métrique de testabilité basée sur les entrées et sorties ainsi que sur la technique du PIE (Propagation, Infection and Execution) pour analyser la testabilité logicielle. Selon Baudry [126,127], la testabilité est une notion centrale dans les systèmes orientés objet dans lesquels le flux de contrôle n'est pas hiérarchique, mais diffus dans toute l'architecture.

D'autres définitions caractérisent la testabilité sous l'angle de l'effort de test. Ainsi, ISO [134] définit la testabilité comme un attribut du logiciel représentant l'effort nécessaire pour le valider. Dans ce cadre, certains auteurs ont étudié la testabilité sous l'angle de l'effort de test requis pour concevoir les tests et/ou découvrir les fautes. C'est ainsi que Binder [103] s'intéresse aux différents facteurs impactant, selon lui, la testabilité et l'assimile au niveau de l'effort à investir pour découvrir des fautes dans un logiciel. Gao et Shih [135] ont relié la testabilité à la réduction de l'effort de test.

Bruntink et Van Deursen [101,102] ont investigué les facteurs liés à la testabilité sous l'angle des tests unitaires (effort d'écriture des tests unitaires) en s'inspirant du diagramme en arêtes de poisson de Binder [103]. Ils ont mis en place une démarche empirique permettant (entre autres) de quantifier la testabilité à travers des métriques statiques du code des classes de test unitaire JUnit. Leurs travaux ont grandement inspiré l'évaluation des efforts unitaires dans nos démarches empiriques. C'est ainsi que nous avons adopté une approche similaire dans [136,137] pour analyser le lien entre la cohésion et la testabilité dans les logiciels OO.

La testabilité reste complexe, mais les métriques logicielles peuvent permettre d'en saisir la notion [101] dans le sens qu'elles permettent aux responsables de projets de planifier, surveiller le processus de test, déterminer les parties critiques du code, et dans certains cas, d'encadrer la révision du code. Cependant, selon Gupta et al.[138], une seule métrique logicielle prise séparément ne suffit pas à donner une vision complète de la testabilité du logiciel. Une manière pratique d'utiliser les métriques pour évaluer la

testabilité (entre autres) et alors de construire des modèles prédictifs à partir de l'information issue de différentes métriques pour identifier les parties du code nécessitant un effort particulier de test. Dans ce chapitre, nous aborderons la testabilité sous l'angle de l'effort de test, plus précisément sous la perspective de l'effort d'écriture et de construction des cas de test unitaire. Nous utiliserons des attributs des classes de test unitaire pour mesurer cet effort.

5.3 Métriques OO et prédiction de l'effort de test unitaire

5.3.1 Objectif

L'objectif de cette partie est d'investiguer empiriquement les liens entre, d'une part, les métriques de design OO issues de la suite de CK et la métrique de taille LOC et, d'autre part, la testabilité des classes du point de vue des tests unitaires. Nous avons analysé 3 grands systèmes logiciels *open source* pour lesquels les suites de test JUnit ont été développées pour une partie des classes des systèmes. L'effort de test a été mesuré par des métriques de code source des classes de test JUnit. Les classes de test associées aux classes logicielles ont été regroupées, en fait, en 2 catégories selon l'effort de test.

Pour étudier les liens entre les métriques OO et l'effort de test unitaire, nous utilisons la régression logistique. La régression logistique univariée est utilisée pour évaluer l'effet individuel de chaque métrique sur l'effort de test unitaire, tandis que la régression logistique multivariée a servi à évaluer l'effet combiné des métriques logicielles. Les performances de ces dernières ont été déterminées par l'analyse des courbes ROC. Les résultats indiquent que: (1) la complexité, la taille, la cohésion et dans une certaine mesure le couplage sont des prédicteurs significatifs de l'effort de test unitaire des classes, et (2) les modèles de régression multivariée basés sur les métriques de design orientées objet peuvent prédire avec précision l'effort de test unitaire des classes.

5.3.2 Métriques et systèmes considérés

5.3.2.1 Les métriques de classes logicielles

Parmi les métriques présentées à la section 3.2.1, nous avons sélectionné 6 métriques de design OO issues de la suite de CK. Il s'agit de CBO, LCOM, DIT, NOC, WMC et RFC. Ces métriques ont reçu une attention particulière de la part des chercheurs et leur lien avec la qualité logicielle a été validé empiriquement [36,39,46,120,139]. Elles sont de plus en plus intégrées dans les environnements de développement. Nous avons choisi, en outre, la métrique de taille LOC comme base de comparaison.

5.3.2.2 Les métriques de test

Pour quantifier les classes test JUnit, nous avons dans cette démarche empirique, choisi les métriques de test TLOC, et TASSERT définies dans la section 3.2.2. Rappelons que ces métriques ont été introduites par Bruntink et Van Deursen [101,102] pour indiquer la taille et le nombre de points de vérifications des suites de test, considérés comme indicateurs de l'effort requis pour tester unitairement les classes. Ces métriques ont d'ailleurs été utilisées dans d'autres travaux [104,136] connexes pour les mêmes buts. Nous supposons que l'effort d'écriture d'une classe test dédiée à une classe logicielle est proportionnel aux deux métriques de test choisies.

5.3.3 Systèmes et statistiques descriptives

Nous avons choisi 3 systèmes *open source* écrits en Java, parmi ceux décrits dans la section 3.1 et les statistiques descriptives des métriques considérées sont données par la table 2 pour toutes les classes (I), et uniquement les classes pour lesquelles des classes test JUnit ont été développées (II). Notons que les classes pour lesquelles les cas de test JUnit ont été développés sont en moyenne plus grandes en taille (en termes de lignes de code), plus couplées (aux autres classes) et ont une complexité cyclomatique plus importante (élevée) quel que soit le système considéré, ce qui en soi est plausible. En effet, les testeurs se focalisent souvent sur ces catégories de classes pour assurer la qualité du produit et minimiser les risques.

Table 2: Statistiques descriptives des métriques des systèmes.

		Toutes les classes I					Classes testées II				
ANT	Variable	Obs.	Min	Max	Moy.	Écart-type	Obs.	Min	Max	Moy.	Écart-type
	CBO	713	0	41	6.59	7.15	111	0	39	10.49	8.57
	DIT	713	0	6	2.20	1.37	111	1	6	2.68	1.34
	NOC	713	0	238	1.52	11.82	111	0	45	0.71	4.33
	LCOM	506	0	3621	76.23	264.04	98	0	3621	155.93	454.19
	RFC	713	0	550	51.20	46.47	111	15	444	78.65	59.83
	WMC	713	0	245	17.10	23.66	111	1	178	31.31	31.11
	LOC	713	1	1252	89.85	130.16	111	3	846	158.64	154.20
	Variable	Obs.	Min	Max	Moy.	Écart-type	Obs.	Min	Max	Moy.	Écart-type
JFC	CBO	496	0	101	10.07	13.46	230	0	67	15.91	15.30
	DIT	496	0	7	1.96	1.55	230	0	5	2.34	1.23
	NOC	496	0	66	2.28	7.53	230	0	50	1.11	5.22
	LCOM	364	0	13 524	224.02	1128.17	219	0	13 524	350.79	1440.48
	RFC	496	0	677	103.47	142.15	230	7	677	150.26	151.35
	WMC	496	0	470	28.10	44.51	230	0	470	46.08	57.12
	LOC	496	2	2041	137.73	216.12	230	3	2041	231.00	273.02
	Variable	Obs.	Min	Max	Moy.	Écart-type	Obs.	Min	Max	Moy.	Écart-type
POI	CBO	1540	0	168	6.41	10.38	372	0	111	10.26	13.24
	DIT	1540	0	7	1.92	1.31	372	0	5	2.16	1.29
	NOC	1540	0	188	1.40	9.65	372	0	49	0.41	2.97
	LCOM	1105	0	131 982	316.77	4347.67	324	0	10 477	185.44	750.92
	RFC	1540	0	642	44.54	58.18	372	0	642	62.30	65.67
	WMC	1540	0	521	16.30	30.16	372	0	374	28.30	35.73
	LOC	1540	2	3560	88.31	172.82	372	2	1409	141.09	180.72

5.3.4 Analyse de corrélation

La première étape de notre démarche empirique est d'explorer les liens entre les métriques de test et les métriques logicielles par une analyse de corrélations. Nous avons calculé les corrélations de Spearman (exposées dans la section 3.4.1.1) pour chaque paire de métriques <métrique test — métrique logicielle>, pour tous les systèmes. Rappelons que cette corrélation est non paramétrique et fait intervenir les rangs des observations. Une valeur significative (proche de 1 en absolu) est le signe d'un fort lien entre les rangs des variables. Le taux de significativité de la *valeur-p* est fixé, comme d'usage, à 5 %. Les valeurs en gras indiquent les corrélations significatives dans la table 3 qui résume les résultats obtenus. L'analyse de la table 3 nous conduit à grouper les résultats en 3 catégories selon le nombre de liens significatifs avec les métriques de test:

Table 3: Corrélations de Spearman entre métriques de test et métriques logicielles.

	ANT		JFC		POI	
	TASSERT	TLOC	TASSERT	TLOC	TASSERT	TLOC
CBO	0.113	**0.373**	**0.264**	**0.304**	**0.289**	**0.342**
DIT	**-0.207**	0.019	0.042	**0.142**	-0.033	**-0.313**
NOC	-0.002	0.058	**0.235**	0.124	0.087	0.078
LCOM	**0.307**	**0.423**	**0.434**	**0.392**	**0.220**	0.061
RFC	0.041	**0.321**	**0.180**	**0.245**	**0.402**	**0.217**
WMC	**0.314**	**0.540**	**0.450**	**0.446**	**0.408**	**0.387**
LOC	**0.319**	**0.542**	**0.413**	**0.432**	**0.378**	**0.370**

- Les métriques de taille et de complexité cyclomatique, respectivement LOC et WMC, présentent des corrélations particulièrement significatives avec les deux métriques de test TLOC et TASSERT pour tous les 3 systèmes. Les corrélations varient de 0.314 pour WMC vs TASSERT dans le cas d'ANT à 0.542 pour LOC vs TLOC, toujours pour le système ANT. Les meilleures valeurs de corrélations de WMC et LOC avec la métrique TASSERT sont observées pour le système JFC. Les résultats de ces deux métriques logicielles sont similaires à cause du fait que la taille capture la complexité cyclomatique (et inversement). Ce résultat est confirmé dans la table 4 avec des corrélations supérieures à 0.92 observées entre ces deux métriques. La métrique TASSERT est bien capturée par WMC.
- Les métriques de couplage CBO, RFC et de cohésion LCOM sont aussi fortement corrélées aux 2 métriques de test pour presque tous les systèmes sauf dans un cas pour chaque métrique. Les métriques RFC et CBO ne sont pas corrélées à TASSERT dans le système ANT et LCOM n'est pas corrélée à la taille de la suite de test (TLOC) dans POI. Ces absences de corrélations pourraient s'expliquer par les spécificités des systèmes et les stratégies de test mises en place par les développeurs. Notons que ces métriques sont significativement corrélées entre elles et avec la taille (LOC) et la complexité (WMC) selon la table 4.
- Pour l'héritage, les liens sont plus faibles. La métrique NOC présente une seule corrélation significative avec TASSERT pour le système JFC. De même, le lien entre DIT, l'autre métrique d'héritage, et l'effort mesuré avec TLOC et TASSERT, semble dépendre du système considéré. La métrique DIT était pressentie pour être très corrélée à l'effort de test (du moins à la taille de la suite: TLOC) dans la mesure où on peut s'attendre qu'à chaque fois que l'on hérite d'une méthode, on devrait la re-tester dans toutes les sous-

classes. Ce résultat peut, cependant, s'expliquer par les stratégies de test adoptées par les développeurs. D'autre part, la faible utilisation de l'héritage dans certains cas, et la granularité unitaire (de classe) considérée peuvent expliquer l'absence de corrélation. À noter, aussi, les corrélations non significatives entre les métriques d'héritage (particulièrement NOC) et les métriques LOC et WMC dans la table 4.

Table 4: Corrélations de Spearman entre métriques logicielles.

		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
ANT	CBO	1	0.185	0.029	0.652	0.832	0.818	0.873
	DIT		1	-0.076	-0.029	0.494	0.115	0.113
	NOC			1	0.31	0.066	0.14	0.078
	LCOM				1	0.67	0.819	0.78
	RFC					1	0.782	0.812
	WMC						1	0.96
	LOC							1
		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
JFC	CBO	1	0.575	0.276	0.644	0.862	0.693	0.792
	DIT		1	0.071	0.349	0.762	0.291	0.376
	NOC			1	0.336	0.268	0.296	0.294
	LCOM				1	0.549	0.757	0.74
	RFC					1	0.64	0.742
	WMC						1	0.966
	LOC							1
		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
POI	CBO	1	-0,209	0,113	0,259	0,626	0,511	0,556
	DIT		1	-0,122	0,093	0,218	-0,114	-0,108
	NOC			1	0,092	0,069	0,060	0,081
	LCOM				1	0,518	0,543	0,534
	RFC					1	0,714	0,725
	WMC						1	0,923
	LOC							1

5.3.5 Modèles logistiques d'évaluation de l'effort de test

Cette section présente l'analyse empirique des effets individuels et combinés des métriques OO sur les métriques de testabilité retenues en termes d'effort de test unitaire. Pour se faire, nous avons eu recourt à la régression logistique univariée et multivariée. Pour obtenir la variable binaire dépendante, nous avons regroupé les variables TASSERT et TLOC selon la moyenne en deux catégories:

- La catégorie 1 regroupe les classes test JUnit pour lesquelles les deux conditions suivantes sont vérifiées: (1) la classe JUnit contient un grand nombre de lignes de code (son TLOC ≥ à la moyenne des observations), et (2) la classe JUnit contient un grand nombre de

points de vérification (TASSERT ≥ moyenne des observations). Nous affectons la valeur 1 aux classes de cette catégorie.

- La catégorie 2 à laquelle nous affectons la valeur 0 regroupe toutes les autres classes JUnit restantes.

Les classes logicielles associées aux classes JUnit de la catégorie 1 ont requis un effort de test relativement plus important en termes d'écriture de code de test et de points de vérifications que les classes logicielles associées à la catégorie 2. La table 5 suivante montre la distribution des classes logicielles suite à cette classification. Nous constatons que pour ces systèmes, et selon notre classification, entre 35 et 39 % des classes logicielles nécessitent un effort de test relativement plus élevé.

Table 5: Distribution des classes logicielles.

	1	0
ANT	38.74 %	61.26 %
JFC	37.17 %	62.83 %
POI	35,04 %	64,96 %

Nous avons construit des modèles logistiques univariés sur les données analysées et avons déterminé les coefficients R^2 et b ainsi que leur *valeur-p* pour comprendre les effets de chaque métrique logicielle prise séparément sur l'effort de test. Les coefficients sont significatifs si leur *valeur-p* est plus petite que le seuil α de 5 %. Nous avons par la suite évalué les performances de prédiction des modèles par le calcul d'AUC défini dans la section 3.4.1.3. Rappelons que l'analyse de l'AUC est moins subjective que le choix d'un point de césure que nécessitent les matrices de confusion, et qu'une valeur > 0.70 de l'AUC est signe que le modèle est un bon prédicteur [113].

La table 6 présente les résultats de la régression logistique univariée obtenus pour les 3 systèmes considérés. Les métriques LOC et WMC ont produit des modèles significatifs pour tous les 3 systèmes avec des R^2 significatifs (allant de 26.96 % pour WMC/JFC à 10.94 % pour LOC/POI) avec des effets significatifs sur l'effort de test (coefficients b significatifs) et de bons modèles prédictifs pour ANT et JFC. Ce résultat était prévisible pour LOC et WMC dans la mesure où ces métriques ont montré de fortes corrélations avec les

deux métriques de test pour tous les 3 systèmes dans la section précédente. Pour LCOM, le coefficient b n'est pas significatif dans le cas d'ANT et de POI, et ces résultats pourraient s'expliquer en partie par la différence entre les tailles des observations. LCOM ne pouvant être calculée pour toutes les classes logicielles (cela est dû à la définition même de la métrique), les observations correspondantes ont dû être éliminées dans la construction des modèles logistiques de LCOM.

La métrique de couplage CBO a des effets significatifs sur l'effort de test. En effet, les coefficients R^2 et b sont significatifs pour tous les systèmes. Cependant, les modèles issus du couplage ne sont pas de bons prédicteurs. Toutes les AUC sont plus petites que 0.7. RFC, qui montrait une corrélation importante avec les métriques TLOC et TASSERT, échoue à bâtir un modèle logistique significatif capable de discerner les classes logicielles nécessitant un effort de test relativement élevé de celles nécessitant un effort de test relativement plus faible.

Pour le système POI, aucune métrique ne fournit un bon prédicteur. En effet, toutes les AUC obtenues des modèles sont plus petites que 0.70.

Table 6: Régression logistique univariée.

		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
ANT	R^2	8.67 %	0.31 %	0.20 %	14.55 %	3.86 %	23.94 %	25.29 %
	2Log	0.007	0.617	0.686	0.001	0.074	< 0,0001	< 0,0001
	b	0.296	-0.054	-0.048	0.946	0.199	0.582	0.596
	valeur-p	0.009	0.618	0.707	0.015	0.091	< 0,0001	< 0,0001
	AUC	0.67	0.549	0.551	**0.73**	0.656	**0.798**	**0.793**
		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
JFC	R^2	11.14 %	0.46 %	5.13 %	13.49 %	0.69 %	26.96 %	13.49 %
	2Log	< 0,0001	0.384	0.003	< 0,0001	0.285	< 0,0001	< 0,0001
	b	0.341	0.066	0.344	1.156	0.081	0.870	1.156
	valeur-p	< 0,0001	0.384	0.045	0.002	0.283	< 0,0001	0.002
	AUC	0.663	0.547	0.598	**0.746**	0.611	**0.79**	**0.746**
		CBO	DIT	NOC	LCOM	RFC	WMC	LOC
POI	R^2	12.94 %	11.23 %	0.00 %	4.82 %	10.29 %	15.26 %	10.94 %
	2Log	< 0,0001	< 0,0001	0.973	0.001	< 0,0001	< 0,0001	< 0,0001
	b	0.420	-0.361	-0.002	0.343	0.409	0.552	0.414
	valeur-p	< 0,0001	< 0,0001	0.973	0.010	< 0,0001	< 0,0001	< 0,0001
	AUC	0.682	0.664	0.533	0.529	0.623	0.686	0.684

L'effet combiné des métriques sur l'effort de test a été analysé par la construction de modèles logistiques multivariés pour chaque système. L'analyse a été effectuée avec et sans la métrique de base de comparaison LOC, afin de voir les effets combinés des métriques de design objet ensemble. Comme dans la régression logistique univariée, nous avons déterminé les coefficients R^2, b ainsi que leurs *valeurs-p*. Nous avons, ensuite, testé les performances des 6 modèles par la détermination de l'AUC de chacun. La table 7 résume les résultats de cette analyse. Pour la régression logistique multivariée, les classes logicielles pour lesquelles la métrique LCOM ne peut être calculée sont exclues des données qui ont servi à bâtir le modèle.

Dans la table 7, nous remarquons une augmentation notable de la significativité et des performances des modèles multivariés comparativement aux modèles univariés. Pour le système ANT, l'amélioration du R^2 passe de 25.9 % à 34.73 % alors que l'AUC augmente de 5 points pour s'établir à 84.48. Nous remarquons pour ce système, que LOC est la seule métrique à avoir un apport (b = 1.66) significatif (valeur-p = 0.022 <5%).

Table 7: Régression logistique multivariée.

	Avec LOC						Sans LOC					
	ANT		JFC		POI		ANT		JFC		POI	
R^2	34.73 %		38.23 %		27.31 %		25.69 %		37.32 %		27.27 %	
2log	0		< 0,0001		< 0,0001		0.002		< 0,0001		< 0,0001	
	b	p-value	b	p-value	b	p-value	b	p-value	b	p-value	b	p-value
CBO	-0.382	0.206	0.218	0.311	0,191	0,217	-0.209	0.432	0.152	0.459	0,196	0,203
DIT	0.271	0.145	0.162	0.198	-0,307	0,000	0.068	0.677	0.188	0.129	-0,304	0,000
NOC	-0.065	0.704	0.095	0.46	0,052	0,465	-0.108	0.591	0.123	0.335	-0,307	0,109
LCOM	0.545	0.296	-1.109	0	-0,305	0,105	0.105	0.798	-1.036	0	0,051	0,472
RFC	-0.962	0.066	-0.617	0.003	0,151	0,513	0.923	0.008	-0.682	0.001	0,155	0,501
WMC	-0.226	0.723	2.333	0	0,392	0,180	-0.351	0.389	1.633	< 0,0001	0,466	0,013
LOC	1.663	0.022	-0.698	0.156	0,081	0,746						
AUC	0.848		0.834		0,773		0.78		0.832		0,772	

Pour JFC et POI, l'influence de LOC est moins importante. De l'analyse avec LOC à l'analyse sans LOC, le R^2 perd 0.1 % de point pour s'établir à 37.32 % et l'AUC perd de 0.02 points. Dans le cas de JFC, les pertes en performance et en significativité sont aussi similaires (0.04 % pour R^2 et 0.01 point pour AUC.). En résumé, l'analyse multivariée montre que la combinaison des métriques OO donne de meilleurs modèles en termes de significativité et de performance que si elles sont prises individuellement.

5.4 Qi et prédiction de l'effort de test unitaire

5.4.1 Objectif

Après l'exploration des liens entre les métriques OO et l'effort de test, nous nous intéressons aux liens entre les indicateurs de qualité Qi et la testabilité du point de vue de l'effort de test. Nous avons gardé la métrique LOC comme base de comparaison. Nous avons mené les travaux d'investigations en 2 étapes. Dans un premier temps, nous avons étudié le lien entre les Qi et la testabilité. Nous évaluons cette fois la testabilité en utilisant différentes métriques de test unitaire. L'analyse des liens est faite à travers une étude de corrélation de Spearman et l'étude sur 2 systèmes *open source* écrits en JAVA, pour lesquels les tests unitaires JUnit ont été développés. Dans un second temps, nous avons approché la problématique de la testabilité sous l'angle de l'effort de test requis pour vérifier unitairement les classes logicielles. Nous avons pour ce faire, étendu les métriques des suites de test en ajoutant 2 nouvelles métriques pour explorer d'autres dimensions (que nous pensions) non capturées par les deux premières proposées par Bruntink et Van Deursen [89,93]. Nous avons, aussi étendu le nombre de systèmes analysés par rapport à l'expérimentation de la section précédente en incluant 3 systèmes *open source* supplémentaires pour avoir un échantillon encore plus significatif. La régression logistique univariée nous a permis de mesurer, après analyse des performances ROC, le lien des métriques avec l'effort de test unitaire. Les résultats montrent l'existence de liens significatifs entre les Qi et la testabilité vue sous l'angle de l'effort (d'écriture) de test. Ce lien est comparable (dans certains cas meilleur) à celui obtenu avec la métrique LOC.

5.4.2 Métriques et systèmes sélectionnés

L'analyse du code source des classes JUnit nous a conduits à choisir trois métriques de test pour quantifier les classes JUnit. En plus de TLOC et TASSERT utilisées dans l'étude précédente, THEFFORT a été incluse (présentée dans la section 3.2.2) pour capturer des dimensions des classes JUnit qui ne sont pas nécessairement saisies par les métriques TLOC et TASSERT. Il s'agit, en effet, de l'effort requis pour comprendre et implémenter une classe en général. Nous avons sélectionné deux systèmes ANT et JFC parmi ceux présentés à la

section 3.1 pour leur grande taille et leurs conceptions différentes (issus de projets différents, et implémentant des fonctionnalités très différentes). Ces systèmes ont, par ailleurs, reçu une attention particulière de la part de la communauté des chercheurs en génie logiciel. Les indicateurs de qualité intrinsèque (Qi^*) ont été inclus dans l'analyse pour comprendre l'effet du flux de contrôle que capturent les Qi sur la testabilité (Qi^* ne capturent pas le flux de contrôle).

Sur le plan des données, nous avons étendu les systèmes analysés dans la deuxième partie des investigations pour les porter à 5: ANT, JFC, JODA, IO et LUCENE. De même, et à des fins d'exploration, nous avons ajouté dans la suite des métriques de test précédentes, la métrique THDIFF présentée à la section 3.2.2.

5.4.3 Statistiques descriptives des systèmes sélectionnés

La table 8 suivante donne certaines caractéristiques des systèmes étudiés permettant de mieux interpréter les résultats expérimentaux qui suivront. Il faut, cependant, noter qu'à cause du calcul des Qi, certaines classes internes (à d'autres classes) voient leur Qi intégré dans le calcul de leurs classes englobantes, ce qui induit des différences entre les observations sur le nombre de classes testées pour JFC (par rapport à l'étude précédente).

Table 8: Statistiques descriptives des 5 systèmes.

	#CL	#ATT	#MET	#LOC	#CT	#LT
ANT	713	2491	5365	64 062	111 (15.6%)	17 609 (27.5%)
JFC	496	1550	5763	68 312	226 (45.6%)	53 115 (77.8%)
JODA	225	872	3605	31 591	76 (33.8%)	17 624 (55.8%)
IO	104	278	793	7631	66 (63.5%)	6326 (82.9%)
LUCENE	659	1793	4397	56 902	114 (17.3%)	22 098 (38.8%)

Ces statistiques montrent que le nombre de classes varie d'un système à un autre, passant de 713 pour ANT à 104 pour IO comme indiqué par la colonne #CL. Nous remarquons aussi qu'avec un nombre de classes plus petit (496), JFC compte plus de méthodes et de lignes de code qu'ANT et LUCENE. LUCENE est similaire à ANT en termes de taille (nombre de lignes de code et nombre de classes). Le système IO est le plus petit des 5 logiciels étudiés. Du côté des tests, la table 8 indique que toutes les classes ne sont pas explicitement testées et que par ailleurs, le nombre des classes explicitement testées varie grandement d'un système à un autre. Selon la colonne #CT, donnant le nombre et le

pourcentage de classes logicielles explicitement testées, 63.5% et 45.6% des classes ont été explicitement couvertes par les tests JUnit développés dans les systèmes IO et JFC, tandis que ANT n'est testé (explicitement) qu'à la hauteur de 15.6% de ses classes.

Nous avions remarqué dans l'étude précédente que les classes testées sont en général plus grandes en termes de lignes de code que les classes non testées. Ce constat nous a poussés à affiner l'analyse en calculant le rapport en termes de lignes de code. La colonne #LT donne le nombre total de lignes de code des classes testées et son ratio avec le nombre total de lignes de code du système. Ainsi, nous voyons que pour JFC par exemple, 45.6% des classes testées représentent en fait 77.8% de lignes de code et que pour IO les 63.5% des classes testées contiennent 82.9% des lignes de code du système. Tous les systèmes affichent cette tendance: Le #LT est plus élevé que le #CT, ce qui veut dire que les classes testées ont en moyenne plus de lignes de code que les classes non testées. Ce constat avait déjà été observé dans l'expérimentation de la section 5.3 pour les systèmes ANT, JFC et POI.

5.4.4 Étude du lien entre Qi et testabilité des systèmes ANT et JFC

Après avoir sélectionné les systèmes et les métriques, nous avons effectué une analyse de corrélation de Spearman entre Qi, Qi^* et les métriques de test sélectionnées. Le seuil de significativité de la *valeur-p* est de 5%. Pour le calcul de Qi et Qi^*, nous avons fixé le taux de couverture des tests unitaires à 75% pour chacune des méthodes des systèmes analysés. En outre, nous avons évalué différentes valeurs de couverture de test, les résultats obtenus ont été sensiblement les mêmes (de façon relative). La table 8, qui montre les résultats de l'analyse, indique en gras les corrélations significatives. Ces résultats indiquent qu'il existe une corrélation significative entre Qi, Qi* et les métriques de test pour les 2 systèmes.

Table 9: Corrélations de Spearman entre Qi/Qi* et les métriques de test.

	ANT			JFC		
	TLOC	TASSERT	THEEFF	TLOC	TASSERT	THEEFF
Qi	-0.560	-0.326	-0.448	-0.425	-0.365	-0.353
Qi*	-0.586	-0.393	-0.523	-447	-0.458	-0.439

Le signe négatif des corrélations confirme que les métriques logicielles (Qi et Qi^*) et les métriques de test (TLOC, TASSER et THEFF) ont des variations inverses, ce qui correspond aux comportements attendus des métriques Qi et Qi*.

5.4.5 Étude des liens entre Qi et l'effort de test

Nous abordons dans cette section la deuxième partie de l'exploration de la testabilité sous l'angle de l'effort de test requis. Nous avons analysé par composantes principales (ACP) les 4 métriques de test sélectionnées plus tôt et servant à évaluer les classes JUnit. L'objectif est d'identifier les recoupements de l'information (s'il y'en a), ainsi que les dimensions orthogonales qu'elles capturent. La table 10 donne dans la première ligne, le cumul de l'information globale (en termes de variance) capturée par les nouvelles composantes principales F1, F2 et F3. Les lignes suivantes montrent la contribution en pourcentage de chaque métrique sur les nouvelles composantes.

Table 10: ACP des 4 métriques de test.

	F1	F2	F3
Cumul	**79.97%**	**92.11%**	**97.67%**
TASSERT	26.17%	23.96%	0.001%
THDIFF	23.51%	29.66%	46.75%
THEFF	24.11%	22.85%	52.96%
TLOC	26.21%	23.54%	0,29%

L'ACP montre que 92.11% de l'information fournie par les métriques de test est répartie sur les 2 premières dimensions orthogonales F1 et F2. La composante F3 qui apporte moins de 5% de la variance totale est ignorée dans l'analyse. En effet, avec plus de 90% de l'information totale capturée par toutes les 4 métriques, ces deux composantes orthogonales suffisent à l'interprétation. Chacune des composantes F1 et F2 peut être représentée par les métriques les plus contributives. Pour la composante F1, les contributions entre les métriques TASSERT et TLOC sont les plus élevées et les plus proches

(26.21 et 26.17). Nous garderons ces deux métriques pour représenter F1. Pour F2, la métrique THDIFF représente à elle seule 29.66% de l'information que la composante cumule, loin devant les autres métriques. THDIFF sera donc la représentante de F2. Nous garderons, suite à cette analyse, les métriques TASSERT, TLOC et THDIFF.

Nous avons effectué une analyse de corrélation de Spearman entre les Qi et les métriques de test retenues. La métrique LOC a servi de base de comparaison. La valeur seuil de la *valeurs-p* est 5%. Les taux de couverture pour les calculs de Qi ont été fixés à 0% par défaut pour toutes les classes afin de mieux refléter la situation d'avant les tests.

Table 11: Corrélations de Spearman entre métriques logicielles et métriques de test.

	ANT			JFC			JODA		
	TASSERT	THDIFF	TLOC	TASSERT	THDIFF	TLOC	TASSERT	THDIFF	TLOC
Qi	**−0.361**	**−0.331**	**−0.553**	**−0.341**	**−0.209**	**−0.415**	**−0.762**	**−0.698**	**−0.805**
LOC	**0.391**	**0.387**	**0.582**	**0.414**	**0.261**	**0.437**	**0.726**	**0.63**	**0.764**
	IO			LUCENE					
	TASSERT	THDIFF	TLOC	TASSERT	THDIFF	TLOC			
Qi	**−0.574**	**−0.550**	**−0.772**	**−0.467**	**−0.306**	**−0.457**			
LOC	**0.641**	**0.585**	**0.827**	**0.495**	**0.316**	**0.47**			

La table 11 montre les résultats des corrélations. Notons que toutes les valeurs de corrélation sont en gras, c-à-d. significatives. Il existe, effectivement, une relation entre les Qi et les métriques de test. Ces résultats viennent confirmer les précédents sur 3 autres systèmes et une métrique de test supplémentaire. Notons que le taux de couverture à 0% a modifié les valeurs des corrélations de Qi pour les systèmes ANT et JFC. Les corrélations se sont légèrement dégradées vis-à-vis de la métrique TLOC mais restent significatives.

Nous avons terminé cette analyse par la construction d'un modèle logistique univarié à partir des données fournies par les Qi, afin d'évaluer à quel point les Qi peuvent capturer l'effort de test combinant les 3 métriques en même temps. Un autre modèle logistique est construit à partir de la métrique de taille LOC pour servir de base de comparaison.

Rappelons que l'évaluation des classes JUnit est assurée par les métriques de test TLOC, TASSERT et THDIFF. Nous avons regroupé les classes logicielles en 2 catégories selon l'effort requis à la création des classes de test JUnit associées. Le regroupement s'est fait en 2 temps, en utilisant les 3 conditions suivantes: Soit C_t une classe test associée à la classe

logicielle C_l . Nous noterons $X(C_t)$ la valeur de la métrique X pour la classe test C_t et $\bar{X}$ la valeur moyenne de la métrique X pour l'ensemble des classes C_t.

- Condition1: $\mathrm{TLOC}(\mathrm{C_t}) \geq \overline{\mathrm{TLOC}}$
- Condition2: $\mathrm{TASSERT}(\mathrm{C_t}) \geq \overline{\mathrm{TASSERT}}$
- Condition3: $\mathrm{THDIFF}(\mathrm{C_t}) \geq \overline{\mathrm{THDIFF}}$

Dans un premier temps, nous avons regroupé les classes en 4 catégories selon les règles suivantes:

- La catégorie 4 regroupe l'ensemble des classes C_l telles que leurs C_t vérifient les 3 conditions.
- La catégorie 3 regroupe l'ensemble des classes C_l telles que leurs C_t vérifient 2 des 3 conditions.
- La catégorie 2 regroupe l'ensemble des classes C_l telles que leurs C_t vérifient au moins 1 des 3 conditions.
- Et la catégorie 1 regroupe l'ensemble des classes C_l telles que leurs C_t ne vérifient aucune des 3 conditions.

Nous avons ensuite regroupé les catégories 4 et 3 pour former le groupe « effort élevé ». Il s'agit de la modalité 1 dont les observations montrent 2 ou 3 valeurs des métriques de test au-dessus de la moyenne. Les catégories 2 et 1 se retrouvent dans le groupe «effort faible ». Il s'agit de la modalité 0 dont les observations ont au plus 1 valeur de métrique de test au-dessus de la moyenne. Nous avons résumé les statistiques de la distribution des classes logicielles selon ce regroupement dans la table 12.

Table 12: Distribution des classes selon "effort élevé"(1) et "effort faible"(0).

	ANT	JFC	JODA	IO	LUCENE
mod-1	33.30%	31.40%	32.90%	30.30%	29%
mod-0	66.70%	68.60%	67.10%	69.70%	71%

Nous remarquons que pour tous les systèmes, environ 30% des classes requièrent un effort élevé selon notre regroupement.

Nous voulons, pour les métriques Qi et LOC, vérifier les hypothèses suivantes:

- H0: Une classe avec une grande valeur de Qi (faible valeur de LOC) ne requiert pas un effort de test plus élevé qu'une classe à faible valeur de Qi (grande valeur de LOC).
- H1: Une classe avec une grande valeur de Qi (faible valeur de LOC) requiert un effort de test plus élevé qu'une classe à faible valeur de Qi (grande valeur de LOC).

L'application de la régression logistique univariée nous donne les résultats résumés dans la table 13. Les performances des modèles ont été évaluées par le calcul d'aire sous la courbe ROC: l'AUC.

Table 13: Régressions logistiques univariées.

		ANT	JFC	JODA	IO	LUCENE
Qi	R^2	0.365	0.16	0.214	0.488	0.177
	2Log	<0.0001	<0.0001	0	<0.0001	0
	b	−0.787	−0.478	−0.838	−1.066	−0.431
	p-value	<0.0001	<0.0001	0.022	<0.0001	0
	AUC	**0.81**	**0.72**	**0.85**	**0.89**	**0.7**
LOC	R^2	0.255	0.19%	0.192	0.593	0.152
	2Log	<0.0001	<0.0001	0.001	<0.0001	0
	b	0.589	0.545	0.562	3.022	0.605
	p-value	<0.0001	<0.0001	0.006	0	0.011
	AUC	**0.8**	**0.75**	**0.8**	**0.91**	0.67

Toutes les *valeurs-p* de la table 13 sont en deçà de 5%, ce qui veut dire que les coefficients b sont significatifs pour tous les systèmes. En d'autres termes, les apports des variables Qi et LOC sont significatifs dans les modèles logistiques des 5 systèmes logiciels choisis. Les valeurs du 2log < 5% montrent, par ailleurs, que les modèles sont significativement différents du modèle de base, en témoigne le coefficient R^2 allant de 48.8% pour IO à 16% pour JFC pour les indicateurs de qualité, et 59.3% avec IO à 15.2% avec LUCENE pour la métrique LOC. De son côté, la valeur du AUC montre que seul le modèle logistique construit à partir du LOC sur les données de LUCENE ne constitue pas un bon prédicteur avec une valeur de 0.67 < 0.7.

Même significatives, les performances et la significativité des modèles varient d'un système à l'autre. Les modèles issus du système IO semblent être les plus significatifs et performants avec les coefficients R^2, b (normalisés) et les AUC les plus élevés. À noter que

ce système montrait déjà le taux (ratio) de lignes de code testées le plus élevé (82.9%) dans les statistiques descriptives (table 8).

Du point de vue des métriques Qi et LOC, les résultats sont comparables. Cependant, les modèles issus de Qi sont de bons prédicteurs (AUC > 0.7) pour tous les 5 systèmes, ce qui n'est pas le cas de LOC pour LUCENE. L'autre avantage du modèle des Qi étant la possibilité d'intégrer les informations sur le taux de couverture des classes au fur et à mesure de l'évolution du processus de test, pour affiner les mesures.

Au regard des résultats obtenus, nous pouvons raisonnablement rejeter l'hypothèse H0 et garder l'hypothèse H1.

5.5 Conclusion et discussion

Nous avons étudié le lien entre les métriques logicielles, les Qi et la testabilité sous l'angle de l'effort de test en deux étapes.

Dans la première partie de ce chapitre, nous avons analysé les relations entre les métriques OO et la testabilité des classes en termes d'efforts requis pour le test unitaire. Nous avons pour cela, effectué une analyse empirique utilisant des données collectées sur 3 grands systèmes *open source* pour lesquels des suites de test JUnit ont été développées. Deux métriques ont été utilisées pour capturer la testabilité des classes, et quantifier les suites de test JUnit correspondantes. Les classes ont ensuite été groupées en 2 catégories selon l'effort de test qu'elles requerraient: fort et faible. Nous avons ensuite évalué l'effet individuel et l'effet combiné des métriques OO grâce aux régressions logistiques univariée et multivariée. Les performances des modèles ont été évaluées par la courbe ROC. Les résultats montrent d'une part, qu'il y'a un lien fort entre les métriques OO et la testabilité telle que mesurée et que d'autre part, les régressions multivariées permettent de prédire avec plus d'exactitude l'effort de test unitaire des classes logicielles. Ce travail a fait l'objet d'une publication [57].

Dans la seconde partie, nous avons inclus les Qi dans les analyses et avons élargi l'ensemble des métriques de test utilisées dans l'investigation précédente en ajoutant 2

nouvelles métriques. Nous avons d'abord effectué une étude de corrélations et avons montré le lien significatif entre les indicateurs de qualité et la testabilité des classes. Nous avons, ensuite, étudié l'information capturée par les 4 métriques de test grâce à l'ACP. Puis, nous avons réduit le nombre de métriques de test à 3, suite à la mise en évidence du recoupement de l'information par l'ACP. Finalement, nous avons regroupé les classes logicielles en 2 catégories selon la moyenne des métriques de test, pour construire un modèle logistique binaire, en prenant la métrique TLOC comme base de comparaison. Les résultats de la régression logistique confirment l'existence des liens entre les indicateurs de qualité et l'effort de test comparables à ceux existant avec la taille (LOC). Ces recherches ont fait l'objet de 2 publications dans [122,123].

Les conclusions de ces investigations se basent sur des échantillons de classes relativement larges. Elles ne peuvent, cependant, pas être généralisées car plusieurs limitations peuvent constituer des risques pour la validité de la démarche. Parmi ces limitations, on peut noter la technique d'appariement manuel décrite à la section 3.3.1. Pour certaines classes, les suites de test sont réutilisées par d'autres classes ce qui n'en fait pas des classes tests dédiées à une classe logicielle spécifique, comme dans l'esprit des tests unitaires. Pour d'autres classes tests JUnit, nous n'avons pas pu établir de correspondance avec des classes logicielles malgré nos efforts d'appariement. Ces quelques classes et l'effort qu'elles ont requis sont ignorés durant cette investigation. Cet aspect peut biaiser la répartition de l'effort de test. Le biais pourrait être réduit dans un environnement contrôlé, respectant strictement les protocoles JUnit. Nous avons aussi noté, d'une part, que dans tous les systèmes analysés, seulement certaines classes logicielles ont été unitairement testées. La proportion des classes logicielles pour lesquelles des tests JUnit ont été explicitement écrits diffère d'un système à un autre et peut éventuellement influencer les résultats. D'autre part, plusieurs classes logicielles pour lesquelles les tests JUnit ont été développés, sont partiellement testées. Autrement dit, dans les classes tests développées, toutes les méthodes ne sont pas testées, et ceci pour plusieurs classes. Il serait intéressant d'explorer des systèmes pour lesquels la couverture de test des méthodes et des classes est maximale.

Dans plusieurs cas, les testeurs semblent se contenter de couvrir uniquement les méthodes les plus complexes. Ce biais semble fortement lié à la stratégie et au style adoptés lors de l'écriture des tests et va constituer l'objet des investigations du chapitre suivant.

CHAPITRE 6. ÉTUDE EMPIRIQUE DES MÉTRIQUES DES TESTS UNITAIRES

6.1 Introduction et Objectif

Nous avons utilisé les métriques de test dans les expérimentations décrites dans les chapitres précédents. Des risques de validité conceptuelle nous ont particulièrement interpelés. Ils concernent: (1) les effets des différents styles adoptés lors de l'écriture des classes tests sur les métriques de test, et (2) la redondance dans l'information que ces métriques de test capturent. Par ailleurs, nous suspectons que ces métriques ne capturent pas toutes les dimensions de l'effort d'écriture des tests. En effet, différentes métriques de test ont été introduites dans la littérature pour évaluer l'effort d'écriture et de constructions des tests unitaires [101,102,104,122,123], sans jamais se poser la question sur la pertinence et le chevauchement des informations qu'elles fournissaient ni sur leur variabilité vis-à-vis des styles d'écriture des classes tests. La redondance et le manque de pertinence de l'information entrainent des difficultés d'interprétation des métriques alors que la sensibilité aux différents styles introduit un biais sur l'information véhiculée. Ce type de biais peut nuire à toute tentative de comparaison de l'effort de test entre différents systèmes.

Nous avons, dans cette investigation, analysé la redondance informationnelle des métriques de test ainsi que l'influence du style adopté lors de l'écriture des tests sur ces métriques dans le but de déceler les métriques de test les moins volatiles qui se recoupent le moins et capturent le plus d'information. Nous avons, par la suite, analysé le lien entre les informations des métriques de test et les attributs logiciels orientés objet. L'expérience a été menée sur un ensemble de 6 systèmes logiciels *open source* pour lesquels certaines classes logicielles ont été explicitement testées.

6.2 Métriques de test unitaire

Nous avons investigué les liens entre les métriques logicielles et l'effort nécessaire à l'écriture des tests unitaires dans les deux sections précédentes. Sur ce thème de recherche, différentes métriques ont été proposées dans la littérature [57,101,102,136,137] et permettent, selon les auteurs, de quantifier différentes perspectives de cet effort. À notre connaissance, aucune étude sur la pertinence de ces métriques n'a été menée à ce jour. Bruntink et Van Deursen [101,102] ont proposé les métriques TLOC et TASSERT (cf. section 3.2.2) pour mesurer 2 perspectives reliées à la taille des tests unitaires en utilisant une version adaptée du diagramme de Binder [103] pour identifier les facteurs de testabilité. Ces deux métriques caractérisent, selon eux, différents facteurs liés au code source des tests, à savoir le nombre de tests nécessaires et l'effort fourni pour développer chaque cas de test. Ils les désignent par facteurs de génération de cas de test et facteurs de conception de cas de test.

Par ailleurs, nous avons analysé dans certains de nos travaux antérieurs le lien entre la cohésion et la testabilité [136,137] sous l'angle de l'effort de test, en utilisant ces deux métriques afin de caractériser l'effort de test unitaire requis. Les résultats montrent une relation significative entre la cohésion et la testabilité. Nous avons également étudié le lien entre le flux de contrôle et la testabilité sous l'angle de l'effort de test comme présenté dans la section précédente. Nous avons utilisé, en plus de TLOC et TASSERT, les métriques de Halstead [105] (THEFF et THDIFF) sur les classes de tests unitaires afin de mesurer l'effort et la difficulté de construction des classes tests. Nos études sur les liens entre métriques de test et métriques logicielles nous ont conduits à analyser de plus près les classes de tests unitaires JUnit. Nous avons alors constaté que les métriques utilisées à ce jour ne capturaient pas certaines dimensions de l'effort relatives à la création de données et de stubs (classes factices simulant les comportements nécessaires aux classes sous test), deux artéfacts importants pour initier les tests unitaires. Nous avons, donc, étendu cet ensemble de métriques de test en rajoutant TDATA et INVOK décrit dans la section 3.2.2 pour saisir ces deux dimensions. De leur côté, Singh et al. [104] ont utilisé les réseaux de neurones pour prédire l'effort de test de régression en termes de lignes de code ajoutées

et changées. Les auteurs ont mesuré l'effort par le nombre de lignes de code, le nombre d'assertions, le nombre de méthodes (TNOO) et le nombre de classes. Les résultats montrent une bonne capacité de prédiction des réseaux de neurones apprenant sur les métriques logicielles.

6.3 Métriques, systèmes logiciels et statistiques descriptives

Après une revue de la littérature, nous avons choisi 3 métriques de test (TLOC, TASSERT et TNOO) que nous avons complété par les métriques TDATA et TNVOK (décrites dans la section 3.2.2) afin de capturer les perspectives de création de données objet et d'utilisation de stubs permettant d'initier les tests unitaires. Les métriques TASSERT et TLOC ont reçu, comme mentionné précédemment, une attention particulière dans la littérature dans la quantification et la caractérisation des classes de test JUnit. Notre ensemble d'étude comporte au final 5 métriques de test: TLOC, TASSERT, TNOO, TDATA, TINVOK (section 3.2.2). Elles capturent, selon leurs auteurs, différentes dimensions de l'effort de test unitaire. Nous avons ainsi regroupé le plus de métriques de test pertinentes, issues de la littérature et de l'étude des classes tests JUnit, afin d'analyser et de comparer l'information qu'elles véhiculent.

Nous avons choisi de caractériser les classes logicielles par les attributs de couplage (CBO), de taille (LOC), et de complexité (WMC) parmi les métriques logicielles décrites à la section 3.2.1. Ces métriques sont aussi connues pour leurs liens empiriques avec la testabilité selon plusieurs études, liens que nous avons pu confirmer par les analyses de la section 5.3. Ces attributs serviront d'angles d'observations des données logicielles. En effet, nous analysons dans ce chapitre la variabilité des métriques de test pour les groupes de classes logicielles (issus de différents systèmes) ayant des valeurs similaires pour chacun de ces 3 attributs.

L'étude est menée sur 6 systèmes logiciels parmi ceux décrits à la section 3.1. Il s'agit d'ANT, JFC, JODA, LUCENE, POI et IVY. Les systèmes sont *open source* et issus de différentes communautés de développeurs et ont été écrits en JAVA. Les statistiques descriptives de la

table 14 donnent une idée sur certaines différences dans les styles d'écriture des classes logicielles et des classes de test.

Table 14: Statistiques descriptives des systèmes.

	#CL	#CT	#LOC	#LT	%CT	%LT	#LTE/#LT
ANT	713	111	64 062	8121	15.60%	27.50%	46.12%
JFC	496	226	68 312	20 657	45.60%	77.80%	38.89%
JODA	225	77	31 591	46 702	34.22%	55.80%	264.17%
LUCENE	659	114	56 902	21 997	17.30%	38.80%	99.54%
POI	1539	404	136 005	41 610	26.25%	43.20%	70.82%
IVY	610	95	50 080	12 531	15.57%	36.00%	69.44%

Nous constatons, en observant la table 14, qu'aucun des systèmes n'est entièrement (explicitement) testé. En effet, le nombre de classes testées (colonne #CT) est systématiquement plus petit que nombre total de classes logicielles (colonne #CL). Les logiciels présentent une répartition des lignes de code par classes très différente d'un système à l'autre. Le système JFC, par exemple, compte beaucoup moins de classes logicielles que ANT, mais contient néanmoins plus de lignes de code (colonne #LOC). Les statistiques montrent, par ailleurs, que les taux de classes logicielles testées et les taux de lignes de code logiciel qu'elles représentent (colonnes %CT et %LT) sont aussi très variables suivant les systèmes. Le pourcentage (dans 1 système) de lignes de code des classes testées est significativement plus élevé que le pourcentage des classes testées.

Pour JFC, par exemple, 45.6% des classes ont été testées et représentent 77.8% des lignes de code du système. Cette tendance est la même pour tous les autres systèmes, ce qui suggère que les classes pour lesquelles des classes tests ont été explicitement développées sont en moyenne plus larges. Le rapport #LTE/#LT donné par la dernière colonne indique le nombre de lignes de test écrites par ligne de code logiciel testée. Il montre que le système JODA a plus de deux lignes de test écrites pour une ligne de code testée. À travers leurs descriptions (cf. la section 3.1), nous constatons que les systèmes choisis traitent de domaines différents. Finalement, avec 1539 classes, POI est le système qui compte le plus de classes logicielles alors que JODA avec 225 classes est le plus petit des 6 systèmes que nous allons analyser.

6.4 Étude de la redondance des métriques de test

Nous avons analysé par composantes principales (ACP cf. section 3.4.1.5) les données de l'ensemble des métriques de test afin de déterminer les dimensions indépendantes qu'elles capturent. L'ACP permet de mettre en évidence les recoupements de l'information véhiculée par différentes variables et de déterminer les dimensions sous-jacentes autour desquelles cette information s'agglutine. Dans notre cas, l'ACP projettera l'information de l'ensemble des 5 métriques de test dans un espace de même dimension et nous suggèrera une base formée de 5 (dans notre cas) composantes principales indépendantes dont l'ordre d'apparition maximise l'information capturée par l'ensemble de nos métriques de test initiales. En considérant uniquement les k premières nouvelles composantes ($k < 5$), nous pouvons garder une grande part de la variabilité (information) des 5 métriques initiales. Nous avons choisi ce critère de variabilité cumulée pour réduire les 5 nouvelles composantes. Nous nous sommes fixés comme objectif, dans cette expérimentation, de ne garder que les n premières composantes qui cumulent 80% de l'information des 5 métriques de test (comme c'est d'usage dans ce type d'analyse). Nous avons effectué la rotation Varimax et avons déterminé les cosinus carrés et les corrélations des métriques de test avec les nouvelles composantes afin d'interpréter correctement et de comprendre l'information capturée par les 5 différentes métriques de test.

Les tables 15 à 20 donnent les résultats des 6 systèmes analysés par cette technique. Les F_i indiquent les nouvelles composantes principales.

Table 15: Résultat ACP pour ANT.

	F1	F2	F3	F4	F5
Variabilité (%)	46.6	33.737	12.049	4.235	3.379
% Cumulé	46.6	80.337	92.386	96.621	100
	ANT (Corrélation)			ANT (Cosinus carré)	
	F1	F2		F1	F2
TINVOK	0.013	0.843		0	0.71
TDATA	0.899	-0.002		0.809	0
TASSERT	0.934	0.102		0.873	0.01
TLOC	0.775	0.515		0.601	0.265
TNOO	0.217	0.837		0.047	0.701

- Pour ANT: Les deux premières composantes de la table 15 couvrent 80.34% de la variabilité globale des 5 métriques. Nous limiterons alors nos interprétations à F1 et F2. Les métriques de test les plus corrélées avec la composante F1 sont TASSERT (0.934), TDATA (0.899) et TLOC (0.775). La composante F2 est, quant à elle, fortement corrélée à TINVOK (0.843) et TNOO (0.837). En termes de classes de test, les deux premières composantes opposent un groupe de classes tests de tailles (TLOC) relativement grandes ayant un grand nombre de points de vérification (TASSERT) et de création de données (TDATA) au groupe de classes tests contenant beaucoup de méthodes (TNOO) et contenant beaucoup d'invocations de méthodes (TINVOK). La forte corrélation positive des métriques TDATA, TASSERT et TLOC avec la première composante F1 montre que ces métriques varient ensemble et dans le même sens pour la plupart des classes de test. Par ailleurs, le fait que F1 et F2 soient indépendantes indique (à travers les métriques de test liées à F2) que les invocations de méthodes et le nombre d'opérations (variant ensemble) varient indépendamment de la taille, du nombre d'assertions et d'objets JAVA créés. Les 80% de variabilité peuvent significativement être représentés par tout couple de métriques issu de l'ensemble {TASSERT, TDATA, TLOC} × {TINVOK, TNOO}.

Table 16:Résultat ACP pour JFC.

	F1	F2	F3	F4	F5
Variabilité (%)	50.152	39.124	6.923	2.771	1.029
% Cumulé	50.15	89.28	96.2	98.97	100
	JFC (Corrélation)			**JFC (Cosinus Carré)**	
	F1	**F2**		**F1**	**F2**
TINVOK	**0.837**	0.358		0.701	0.129
TDATA	0.209	**0.951**		0.044	0.905
TASSERT	0.647	0.694		0.419	0.481
TLOC	**0.746**	0.634		0.556	0.402
TNOO	**0.887**	0.197		0.787	0.039

- Pour JFC: Avec 89.28% de variabilité, nous limiterons les interprétations de l'ACP de JFC aux deux premières composantes F1 et F2 de la table 16. La composante F1 est fortement corrélée aux métriques de test TNOO (0.887), TINVOK (0.837) et TLOC (0.746). Cette composante regroupe les classes tests de tailles (TOC) relativement grandes ayant beaucoup de méthodes (TNOO) dans lesquelles beaucoup d'invocations de méthodes sont effectuées (TINVOK). Les signes positifs des corrélations avec F1 montrent que ces

métriques varient dans le même sens. F2 est fortement représentée par TDATA avec une corrélation de 0.951. Les corrélations relativement élevées de TASSERT et TLOC n'en font pas de bons représentants de F2. En effet, leurs très faibles cosinus carrés (0.481 et 0.402 respectivement) indiquent que les deux métriques sont très éloignées de l'axe de projection (la composante F2). F2 regroupe donc uniquement des classes tests dans lesquelles beaucoup d'objets sont créés. L'indépendance des dimensions F1 et F2 indique que la création d'objets dans les classes tests se fait indépendamment de la taille, du nombre d'opérations et des invocations de méthodes. Les 89.28% de variabilité observée peuvent être significativement être représentées par chaque couple de métriques issu de l'ensemble {TNOO, TINVOK, TLOC} × {TDATA}.

- Pour JODA: Dans la table 17, l'ACP des données issues des 5 métriques de test de JODA montre la première composante F1 cumulant plus de 93.36% de la variance totale. Elle suffit à elle seule, d'après notre critère, à interpréter la variance totale des 5 métriques. Les 5 métriques capturent la même information. Elles sont, en effet, toutes significativement corrélées (avec des valeurs > 0.90) à cette composante. Ce résultat peut s'interpréter, entre autres, comme un certain équilibre dans la plupart des classes tests. L'effort fourni par les développeurs pour l'écriture, la vérification et la création de données est proportionnellement distribué dans les différentes classes de test.

Table 17: Résultats ACP pour JODA.

	F1	F2	F3	F4	F5
Variabilité (%)	93.366	2.949	2.454	0.996	0.235
% Cumulé	93.366	96.315	98.769	99.765	100
	JODA (Corrélation)			JODA (Cosinus carré)	
	F1			F1	
TINVOK	0.952			0.906	
TDATA	0.953			0.908	
TASSERT	0.955			0.911	
TLOC	0.99			0.98	
TNOO	0.982			0.964	

Le ratio entre le nombre de lignes de test écrites et le nombre de lignes de code logiciel testées (#LTE/#LT de la table 14) abonde dans le sens de cette explication. En effet, avec 264.17%, le système JODA présente et de loin, le ratio le plus élevé parmi les systèmes

étudiés, soit 2 lignes de test écrites pour 1 ligne de code logiciel testée. Dans ce système, chacune des métriques de test de l'ensemble {TASSERT, TDATA, TLOC, TINVOK, TNOO} peut représenter significativement la composante F1.

- Pour LUCENE: La table 18 montre que les résultats de l'ACP des données de test de LUCENE suivent la même tendance que ceux de JODA. En effet, les 88.96% de variabilité de l'information issue des métriques de test se concentrent sur la première composante F1, ce qui est suffisant, d'après notre critère, pour limiter nos interprétations à cette composante.

Table 18: Résultat ACP pour LUCENE.

	F1	F2	F3	F4	F5
Variabilité (%)	88.965	6.462	2.144	1.429	1
% Cumulé	88.965	95.427	97.571	99	100
	LUCENE (Corrélation)			LUCENE (Cosinus carré)	
	F1			F1	
TINVOK	0.972			0.945	
TDATA	0.95			0.903	
TASSERT	0.86			0.936	
TLOC	0.967			0.739	
TNOO	0.962			0.925	

Les métriques de test sont toutes significativement corrélées (> 0.86) à cette composante et proches, selon les cosinus carrés (> 0.5), de l'axe de projection. L'effort d'écriture, de vérification et de création de données semble être relativement bien distribué dans les différentes classes de test unitaire. Notons que dans le cas de LUCENE aussi le ratio entre le nombre de lignes de test et le nombre de lignes de code logiciel testées est de 99.54%, soit le deuxième le plus élevé après JODA. Autrement dit, à peu près 1 ligne de test est écrite pour chaque ligne de code testée. À titre de comparaison, les systèmes restants ont un ratio en deçà de 71%. Toute métrique de l'ensemble {TASSERT, TDATA, TLOC, TINVOK, TNOO} peut représenter significativement la composante F1.

- Pour POI: Afin d'interpréter les résultats du plus grand des 6 systèmes, présentés dans la table 19, nous avons besoin de considérer les composantes F1 et F2 pour respecter notre critère et ainsi cumuler au moins 80% de l'information (84.21%). Les métriques de test les plus corrélées avec la première composante F1 sont TDATA (0.896), et TLOC (0.887), tandis que la deuxième composante F2 est fortement corrélée aux métriques de test TINVOK (0.912) et TNOO (0.758).

Table 19: Résultat ACP pour POI.

	F1	F2	F3	F4	F5
Variabilité (%)	47.43	36.784	8.409	5.713	1.664
% Cumulé	47.43	84.214	92.623	98.336	100
	POI (Corrélation)			**POI (Cosinus carré)**	
	F1	**F2**		**F1**	**F2**
TINVOK	0.264	**0.912**		0.069	0.832
TDATA	**0.896**	0.237		0.802	0.056
TASSERT	0.697	0.467		0.486	0.218
TLOC	**0.887**	0.399		0.787	0.159
TNOO	0.476	**0.758**		0.227	0.574

La métrique TASSERT n'apporte pas d'information assez significative dans ces deux dimensions pour être interprétée. En termes de classes de test, les deux premières composantes opposent un groupe de classes tests de tailles (TLOC) relativement grandes avec beaucoup de création de données (TDATA) au groupe de classes tests définissant beaucoup d'opérations (TNOO) et contenant plusieurs invocations de méthodes (TINVOK). Les fortes corrélations positives des métriques TDATA et TLOC avec la première composante F1 montrent que ces métriques varient ensemble et dans le même sens pour la plupart des classes de test. F1 et F2 sont des dimensions indépendantes, ce qui indique (à travers les métriques de test liées à F2) que les invocations de méthodes et le nombre d'opérations (variant dans le même sens) varient indépendamment de la taille, et du nombre d'objets JAVA créés dans le code des classes tests. Les 84.21% de variabilité peuvent significativement être représentés par tout couple de métriques de test appartenant à l'ensemble {TDATA, TLOC} × {TINVOK, TNOO}.

- Pour IVY: Les données de la table 20 révèlent que 89.32% de l'information globale des 5 métriques se concentre sur les deux premières composantes. Elles sont, donc, suffisantes selon notre critère pour interpréter les résultats de l'ACP d'IVY. La composante F1 est significativement représentée par les métriques TASSERT, TNOO et TLOC avec des corrélations respectives de 0.92, 0.78, et 0.792, alors que la composante F2 est fortement liée aux métriques TDATA et TINVOK avec 0.92 et 0.75 de corrélations. L'indépendance de F1 et F2 suggère que les attributs caractérisant ces deux groupes varient indépendamment les uns des autres. Finalement, les 89.32% d'information capturée par les 5 métriques de test peuvent significativement être représentés par chaque couple de métriques

appartenant à l'ensemble {TASSERT, TLOC, TNOO} × {TDATA, TINVOK}.

Table 20: Résultats ACP pour IVY.

	F1	F2	F3	F4	F5
Variabilité (%)	48.525	40.798	6.483	3.782	0.412
% Cumulé	48.525	89.323	95.806	99.588	100
	IVY (Corrélation)			IVY (Cosinus carré)	
	F1	F2		F1	F2
TINVOK	0.529	**0.748**		0.280	0.559
TDATA	0.240	**0.923**		0.058	0.851
TASSERT	**0.923**	0.201		0.853	0.041
TLOC	**0.780**	0.613		0.609	0.375
TNOO	**0.792**	0.462		0.627	0.214

Nous avons résumé, dans la table 21, le résultat de l'ACP des 6 systèmes. Nous avons déterminé les meilleurs sous-ensembles de métriques issus des ensembles représentatifs des composantes principales selon la valeur de la corrélation des métriques avec la composante F1 ou F2.

Table 21: Résumé des résultats de l'ACP.

	F1	F2	Meilleur sous-ensemble
ANT	TASSERT, TDATA, TLOC	TINVOK, TNOO	(TASSERT, TINVOK)
JFC	TNOO, TINVOK, TLOC	TDATA	(TNOO, TDATA)
JODA	TASSERT, TDATA, TLOC, TINVOK, TNOO	X	(TLOC)
LUCENE	TASSERT, TDATA, TLOC, TINVOK, TNOO	X	(TINVOK)
POI	TDATA, TLOC	TINVOK, TNOO	(TDATA, TINVOK)
IVY	TASSERT, TLOC, TNOO	TDATA, TINVOK	(TASSERT, TDATA)

Les différentes analyses suggèrent que plus de 80% de la variabilité est répartie sur (au plus) deux dimensions sous-jacentes indépendantes. Dans la première dimension, revient systématiquement la métrique TLOC. On peut la considérer comme représentante significative de cette première dimension capturée par les métriques de test. Dans la deuxième dimension (dans le cas où elle est nécessaire pour cumuler les 80% de l'information) TINVOK revient le plus souvent, à l'exception de JFC où elle est incluse dans la première dimension. TINVOK est la métrique la plus indépendante de la taille (au vu des systèmes analysés) dans le cas où les informations sont réparties sur deux dimensions. Par conséquent, nous pouvons considérer le couple de métriques (TLOC, TINVOK) comme étant le plus indépendant et capturant le plus d'information issue des données des 5 métriques de test des systèmes analysées. Les deux métriques suffisent donc à relativement bien

caractériser l'information fournie par les 5 métriques de test.

Nous pensons que les variations de représentants pour les axes F1 et F2 (et du meilleur sous-ensemble de métriques dans la table 21) sont liées aux différents styles adoptés par les testeurs et à la sensibilité des métriques de test vis-à-vis de ce style (volatilité). En effet, l'écriture des tests unitaires est empreinte du style d'écriture des développeurs des tests. Nous avons constaté, par exemple, en analysant les classes des tests unitaires, que dans certains cas, les développeurs avaient tendance à regrouper les assertions dans des méthodes utilitaires séparées qu'ils invoquaient à chaque point du code de test où la vérification des résultats est nécessaire. Ce simple regroupement fonctionnel du code biaise le décompte statique des assertions, réalisé par la métrique TASSERT. Ce qui montre que TASSERT (comme probablement les autres métriques de test) est volatile, c-à-d. sensible aux styles adoptés lors de l'écriture des tests. Nous conduisons, dans la section suivante, une investigation qui vise à isoler les métriques de test les moins volatiles parmi TLOC, TASSERT, TDATA, TINVOK et TNOO.

6.5 Étude de la variance des métriques de test

Dans cette section, nous investiguons la variance des métriques de test selon les différents styles d'écriture des classes JUnit. L'objectif est de déterminer les métriques les moins volatiles face aux différents styles adoptés par les développeurs lors de l'écriture et la création des classes de test unitaire. Nous allons, pour ce faire, regrouper les classes logicielles de l'ensemble des (6) systèmes selon les attributs de taille, de couplage, et de complexité. Le regroupement se fera, d'un côté avec la technique du KMEAN (exposée à la section 3.4.2.1) pour les 3 attributs pris ensemble et, de l'autre, par la technique de partitionnement univarié pour chacun des 3 attributs pris séparément. Les 4 processus de classifications (1 KMEAN et 3 partitionnements univariés) vont regrouper les classes logicielles en 5 clusters qui minimisent la variance intra-cluster et maximisent la variance inter-cluster, selon l'attribut ou le groupe d'attributs considérés. Dans chaque cluster, nous nous assurerons que les différents systèmes sont bien représentés par le calcul de la variance qualitative. Finalement, nous analyserons les variances des 5 métriques de test sur

chacun des clusters des classes tests associés aux clusters des classes logicielles. La figure 3 résume la démarche expérimentale.

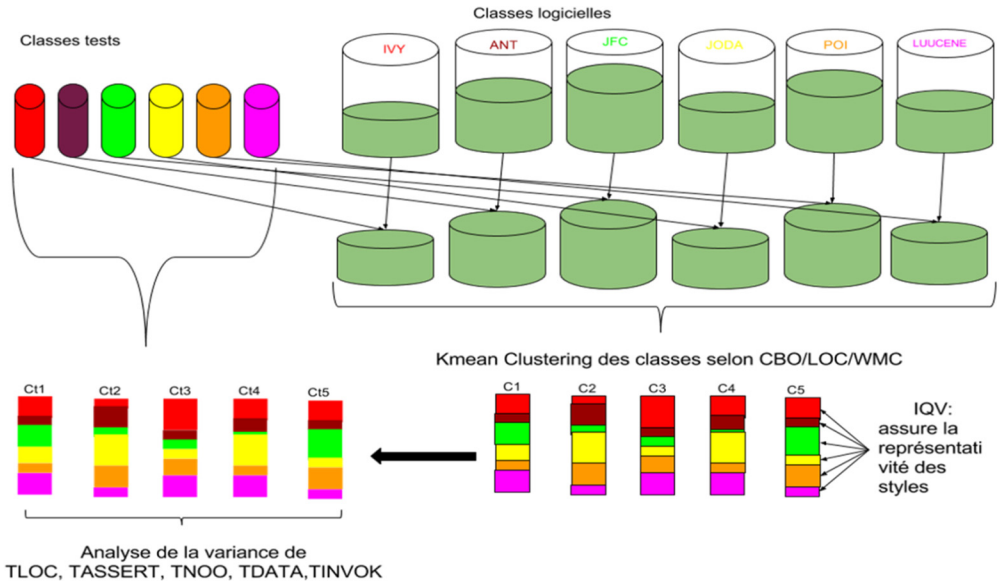


Figure 4: Étude de la variance des métriques de test selon les styles.

6.5.1 Regroupement selon les 3 attributs des classes logicielles

Dans un premier temps, nous avons regroupé les classes logicielles (et leurs classes tests correspondantes) selon les attributs de taille (TLOC), couplage (CBO) et complexité (WMC) pris ensemble avec la technique du KMEAN. Le but étant d'avoir des clusters dans lesquels les variances internes de ces 3 attributs sont relativement peu importantes (attributs comparables). La table 22 résume ainsi, les statistiques descriptives des 3 attributs pour les 1027 classes logicielles et les 5 clusters obtenus après la classification KMEAN sur les données des attributs CBO, LOC et WMC prises ensemble. Le nombre de clusters a été fixé à 5 pour correspondre aux 5 catégories utilisées couramment: très faible, faible, moyen, élevé et très élevé.

Nous avons ordonné de manière croissante les clusters obtenus de 1 à 5 selon leurs barycentres. Nous remarquons que les moyennes des attributs suivent toutes cette

tendance et croissent ainsi du cluster 1 au cluster 5. Notons que le nombre d'observations diminue quand on passe du cluster 1 au cluster 5.

Table 22: Statistiques descriptives des attributs internes des clusters KMEAN.

		6 Systèmes	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Nb. Obs.		1027	587	267	118	38	17
CBO	Min	0	0	0	0	4	3
	Max	111	44	68	57	92	111
	Moy (μ)	12.359	6.058	14.461	24.559	43.026	43.706
	Var (σ^2)	180.84	29.40	100.56	152.01	396.13	831.38
LOC	Min	2	2	124	292	592	1134
	Max	2644	128	288	562	1039	2644
	Moy (μ)	182.375	61.566	192.56	387.11	770.45	1458.24
	Var (σ^2)	58 255.06	1106.71	2333.19	5940.44	18 137.83	165 272.42
WMC	Min	0	0	3	16	10	10
	Max	557	86	98	157	231	557
	Moy (μ)	35.42	13.33	38.69	77.86	137.34	224.29
	Var (σ^2)	2099.52	75.38	195.85	506.93	2372.91	23 911.86
IVQ		0.91	0.867	0.923	0.968	0.886	0.839

Les variances entre les clusters ne sont pas égales et croissent, elles aussi, avec les moyennes des attributs des clusters. Dans cette table, nous avons calculé l'indice de variance qualitative IVQ [140] pour nous assurer que les différents systèmes sont significativement représentés dans nos clusters avant de débuter l'analyse de variance. L'indice IVQ détermine ici la variance des 6 systèmes au sein des groupes. Il est donné par la formule suivante:

$$\mathrm{IVQ} = \frac{k\left(n^2 - \sum_{i=0}^{n}\left(f_i^2\right)\right)}{n^2(k-1)}$$

Où n indique la taille du groupe, k = 6 indique le nombre de systèmes, f_i la fréquence de la modalité i (des classes du *i*-ième système) dans le groupe.

L'indice IVQ varie entre 0 et 1 dans un groupe donné lorsque toutes les classes sont issues du même système. Lorsque l'IVQ est nulle, cela signifie qu'il n'y a pas de diversité. L'indice de variance qualitative indique le pourcentage de différence relative par rapport au maximum de différences dans chaque groupe. Nous remarquons que les indices de variance qualitative sont élevés dans les 5 clusters et sont proches de celui du groupe contenant toutes les classes des 6 systèmes. Les différents systèmes (styles d'écriture) sont donc bien représentés dans les 5 clusters.

Pour les classes tests correspondant aux classes logicielles, les variances des métriques de test sont aussi calculées sur les 5 clusters ainsi que sur le groupe formé de l'ensemble des classes tests. La table 23 donne le minimum, le maximum, la moyenne (μ), la variance (σ^2) et le coefficient de variation (C_v) des 5 métriques de test (pour les 5 clusters), ainsi que du groupe formé de l'ensemble des classes tests. Défini comme l'écart-type reporté à la moyenne, le coefficient de variation C_v ou écart type relatif est moins biaisé par les échelles différentes des métriques de test. Nous l'utiliserons pour comparer les variances entre les différentes métriques des groupes.

Le premier constat que nous pouvons faire à partir de la table 23 est que toutes les moyennes des métriques de test augmentent dans l'ordre des clusters, donc avec les moyennes des 3 attributs des classes logicielles à l'origine de cette classification des classes test. Sachant que pour chaque cluster de classes tests, les classes logicielles correspondantes appartiennent au même cluster (et ont des attributs relativement comparables), de grandes variances observées dans les métriques de test résulteraient en grande partie de leur sensibilité vis-à-vis des différents styles d'écriture présents dans ces clusters. Comme le montre la figure 5, TASSERT a les plus grands coefficients de variation, suivi de TDATA et TNOO. Ces métriques semblent volatiles et biaisées par les différents styles présents dans les clusters.

Table 23: Descriptions et variances des clusters de classes tests selon le KMEAN logiciel.

		Ensemble des classes tests	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Nb. Obs.		1027	587	267	118	38	17
TLOC	Min	6	6	8	10	16	18
	Max	4063	1280	2035	2236	2624	4063
	Moy (μ)	147.63	84.78	146.99	289.71	399.08	779.59
	Var (σ^2)	83 443.90	12 232.29	43 347.72	188 423.24	366 741.81	992 939.77
	Coef. of var ($C_v = \sigma/\mu$)	1.96	1.304	1.416	1.498	1.517	1.278
TASSERT	Min	0	0	0	0	1	2
	Max	1156	329	1058	1014	1156	396
	Moy (μ)	36.77	17.76	36.88	84.70	130.40	149.35
	Var (σ^2)	9284.81	1180.64	7549.79	27 340.33	59 293.98	18 567.99
	Coef. of var ($C_v = \sigma/\mu$)	2.62	1.935	2.356	1.952	1.867	0.912
TDATA	Min	0	0	0	0	0	0
	Max	758	324	482	393	482	758
	Moy (μ)	21.20	11.64	22.31	43.62	56.79	98.94
	Var (σ^2)	2609.12	467.34	1989.79	5721.88	11 285.54	29 750.76
	Coef. of var ($C_v = \sigma/\mu$)	2.41	1.857	2	1.734	1.871	1.743
TINVOK	Min	0	0	2	1	6	2
	Max	516	175	192	401	399	516
	Moy (μ)	35.06	26.284	34.337	60.754	72.868	86.294
	Var(σ^2)	1801.45	731.42	967.25	3781.46	7094.17	13 206.33
	Coef. of var ($C_v = \sigma/\mu$)	1.21	1.029	0.906	1.012	1.156	1.332
TNOO	Min	0	0	1	1	1	3
	Max	242	153	164	242	238	148
	Moy (μ)	10.26	6.19	10.27	21.09	30.74	29.65
	Var (σ^2)	492.24	107.79	314.80	1536.74	2868.93	1292.35
	Coef. of var ($C_v = \sigma/\mu$)	2.16	1.678	1.728	1.859	1.743	1.213

Nous remarquons également que la métrique de test TINVOK a les plus faibles coefficients de variations. Elle est suivie de la métrique de test TLOC. Le calcul de la moyenne des coefficients de variation des métriques de test pour chaque cluster suggère (figure 4) que TINVOK et TLOC sont effectivement les métriques qui varient le moins en moyenne. Pour TLOC et TINVOK, ce résultat peut s'interpréter comme suit: Pour des classes logicielles de tailles, de couplages et de complexités semblables, le nombre de lignes de code et les invocations de méthodes dans les classes tests dédiées varient relativement peu quelque soit les différents styles adoptés par les développeurs lors de l'écriture des tests des 6 systèmes.

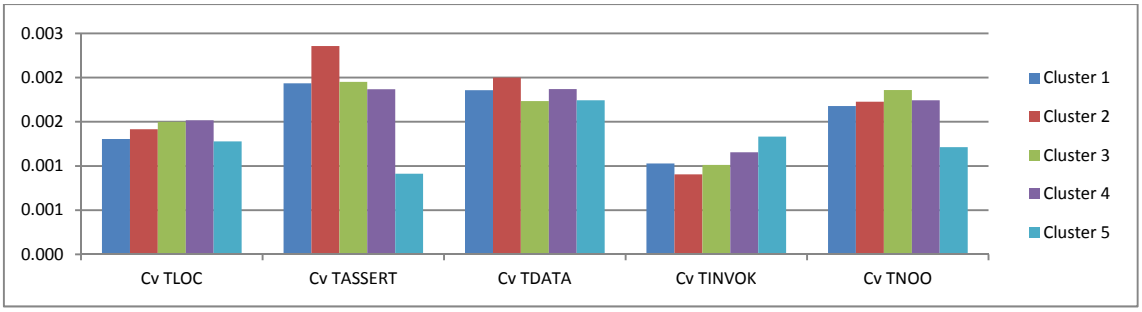


Figure 5: Coefficients de variations (C_v) des métriques de test selon chaque cluster.

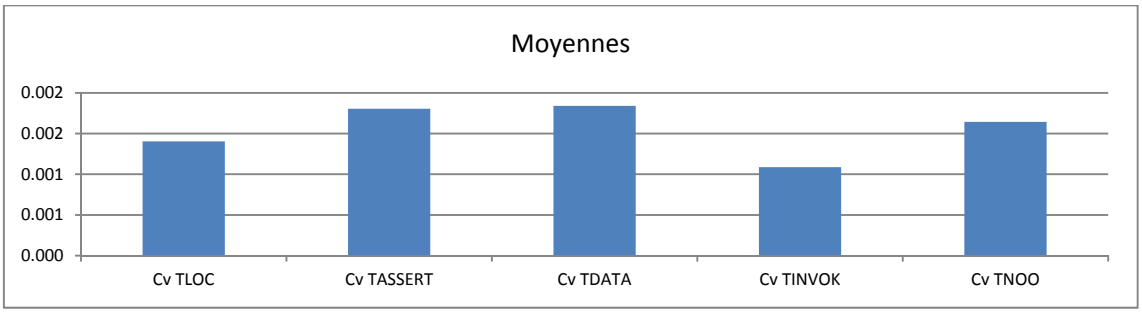


Figure 6: Moyenne des coefficients de variation des métriques de test.

Nous reproduisons le regroupement ainsi que l'analyse de variance dans les sous-sections qui suivent, selon les attributs de taille, de complexité et de couplage pris séparément. L'objectif est de voir si d'autres métriques émergent comme étant les moins volatiles vis-à-vis du style d'écriture des tests unitaires des systèmes pour les attributs logiciels pris séparément.

6.5.2 Regroupement des tests selon la taille (LOC) des classes logicielles

Dans cette section, nous effectuons un partitionnement univarié (cf. section 3.4.2.1) des 1027 classes logicielles selon leur taille. Le processus produit 5 clusters dont les variances sont relativement faibles par rapport aux variances entre les clusters. Au sein de chaque cluster, les classes logicielles ont des tailles relativement comparables. Nous les avons ordonnées selon la moyenne (de LOC) des clusters de la plus faible (cluster 1) à la plus forte (cluster 5) moyenne. Ensuite, nous avons calculé l'IVQ de chaque cluster pour vérifier la représentativité des 6 systèmes dans les 5 clusters. Pour les 5 clusters des classes tests associées aux clusters des classes logicielles, la table 24 décrit les métriques de test, l'indice IVQ et donne aussi les coefficients de variation permettant de les comparer.

Les différents indices de variance qualitative sont assez proches de 1 ce qui montre que les différents systèmes (donc les différents styles adoptés lors de l'écriture des classes de test) sont significativement représentés dans les 5 clusters des classes tests. La plus faible valeur d'IVQ (0.768) se trouve dans le cluster 5. Ce qui peut s'expliquer par la différence de tailles des classes d'un système à l'autre ainsi que la petite taille du cluster 5 (5 observations uniquement).

Table 24: Analyse de la variance 5 clusters uniformément partitionnés selon LOC.

		Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
	Nb. Obs.	707	236	53	26	5
	IVQ	0.877	0.943	0.912	0.895	0.768
TLOC	Min	6	8	23	16	323
	Max	1620	2035	2624	2379	4063
	Moy (μ)	93.919	193.042	406.755	437.577	1344.8
	Var (σ^2)	15 472.165	82 475.244	434 079.053	267 044.936	1 981 534.96
	Coef. of var ($C_v = \sigma/\mu$)	1.324	1.488	1.62	1.181	1.047
TASSERT	Min	0	0	1	1	79
	Max	615	1058	1156	528	329
	Moy (μ)	20.446	49.394	142.283	129.731	146.8
	Var (σ^2)	1679.404	11 552.171	72 871.184	23 666.274	8614.16
	Coef. of var ($C_v = \sigma/\mu$)	2.004	2.176	1.897	1.186	0.632
TDATA	Min	0	0	0	0	9
	Max	482	326	482	323	758
	Moy (μ)	13.105	30.424	56.925	47.654	214.8
	Var (σ^2)	746.744	2898.566	11 869.126	4633.842	75 671.76
	Coef. of var ($C_v = \sigma/\mu$)	2.085	1.77	1.914	1.428	1.281
TINVOK	Min	0	2	1	2	17
	Max	175	327	401	138	516
	Moy (μ)	27.081	44.636	75.604	58.846	157
	Var (σ^2)	725.624	1831.927	8548.843	1605.053	33 416.8
	Coef. of var ($C_v = \sigma/\mu$)	0.995	0.959	1.223	0.681	1.164
TNOO	Min	0	1	1	1	3
	Max	164	156	242	94	148
	Moy (μ)	6.859	13.186	33.396	22.692	42.6
	Var (σ^2)	136.517	575.753	3710.881	552.136	2857.04
	Coef. of var ($C_v = \sigma/\mu$)	1.704	1.82	1.824	1.035	1.255

La table 24 montre, par ailleurs, que les moyennes des différentes métriques croissent avec les clusters, ce qui est plausible. En effet, vu que les clusters sont ordonnés selon les moyennes des tailles des classes logicielles, cela dénote plutôt du lien entre la taille des classes logicielles et les différentes métriques de test. Nous remarquons aussi que les métriques TLOC et TINVOK ont les coefficients de variations les plus faibles dans les clusters, alors que TASSERT et TDATA sont les métriques qui fluctuent le plus au sein des mêmes clusters (figure 6). Nous avons calculé les moyennes des coefficients de variation

des métriques de test sur les 5 clusters et les avons représentées dans la figure 7. Le Graphique confirme que TINVOK et TLOC sont en moyenne moins volatiles que les métriques TDATA, TASSERT et TNOO.

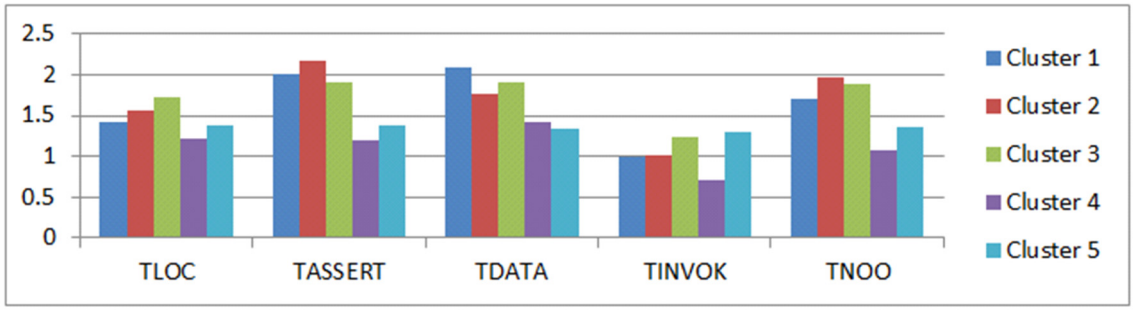


Figure 7: Coefficients de variations (C_v) des métriques par cluster, partitionnement selon LOC.

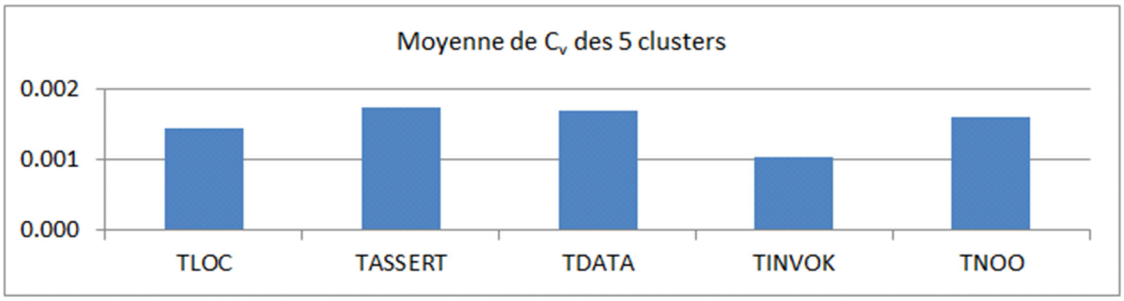


Figure 8: Moyenne des coefficients de variation des métriques de test, partitionnement selon LOC.

6.5.3 Regroupement selon la complexité (WMC) des classes logicielles

Dans cette section, nous effectuons un partitionnement univarié (cf. section 3.4.2.1) des 1027 classes logicielles selon leur complexité en 5 clusters. Au sein de chaque cluster, les classes logicielles ont des complexités cyclomatiques relativement comparables. Nous les avons ordonnées selon les moyennes (de WMC) des clusters de la plus faible (cluster 1) à la plus forte (cluster 5). Les 5 clusters formés par les classes tests associées aux classes logicielles sont décrits dans la table 25. Comme pour le partitionnement selon la métrique LOC, nous avons calculé les statistiques descriptives des métriques de test, l'indice IVQ de chaque cluster ainsi que les coefficients de variations permettant de comparer la volatilité des différentes métriques.

La distribution des observations dans les différents clusters est similaire à celle de LOC, ce qui peut s'expliquer en partie par le lien de corrélation entre le nombre de lignes

de code (LOC) et la complexité cyclomatique (WMC) des classes logicielles. Les différents indices de variance qualitative assez proches de 1 (sauf pour le cluster 5) nous garantissent que les 6 styles sont significativement représentés dans les 4 premiers clusters.

Table 25: Analyse de la variance 5 clusters uniformément partitionnés selon WMC.

		Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
	Nb. Obs.	632	279	93	18	5
	IVQ	0.869	0.941	0.957	0.889	0.672
TLOC	Min	6	8	20	16	408
	Max	1358	1745	2624	2280	4063
	Moy (μ)	83.97	175.68	375.18	473.28	1224.60
	Var (σ^2)	12 046.34	55 144.96	342 525.70	301 335.20	2 022 982.64
	Coef. of var ($C_v = \sigma/\mu$)	1.307	1.337	1.56	1.16	1.161
TASSERT	Min	0	0	0	1	104
	Max	528	615	1156	832	391
	Moy (μ)	17.57	42.33	120.12	143.67	217.4
	Var (σ^2)	1346.74	5860.03	53 999.37	38 933	14 041.84
	Coef. of var ($C_v = \sigma/\mu$)	2.09	1.81	1.94	1.37	0.55
TDATA	Min	0	0	0	1	19
	Max	161	482	482	323	758
	Moy (μ)	10.90	27.341	54.66	59	223.2
	Var (σ^2)	261.22	2712.48	8866.05	8677.78	73 098.16
	Coef. of var ($C_v = \sigma/\mu$)	1.483	1.905	1.723	1.579	1.211
TINVOK	Min	0	1	1	6	82
	Max	154	216	401	323	516
	Moy (μ)	25.67	39.88	66.95	82.39	188.40
	Var (σ^2)	653.03	1283.52	5772.44	5088.90	27 243.84
	Coef. of var ($C_v = \sigma/\mu$)	0.995	0.898	1.135	0.866	0.876
TNOO	Min	0	1	1	1	11
	Max	82	164	242	188	148
	Moy (μ)	5.76	12.92	26.55	33.5	44
	Var (σ^2)	58.43	481.32	2465.50	2028.81	2726
	Coef. of var ($C_v = \sigma/\mu$)	1.33	1.70	1.87	1.35	1.19

La table 25, comme la table 24 précédente, montre que les moyennes des différentes métriques croissent avec les clusters. Ce fait traduit le lien entre la complexité des classes logicielles et les métriques de test. Dans le cas de ce partitionnement aussi, les métriques TLOC et TINVOK ont les coefficients de variations les plus faibles dans les clusters, alors que TASSERT et TDATA sont les métriques qui fluctuent le plus au sein de ces mêmes clusters (figure 9). Peu de fluctuations sont observées dans le cluster 5 principalement à cause de sa petite taille (5 observations). La figure 10 montre la moyenne des coefficients de variation des métriques de test à travers tous les clusters et confirme que TINVOK puis TLOC sont les moins volatiles.

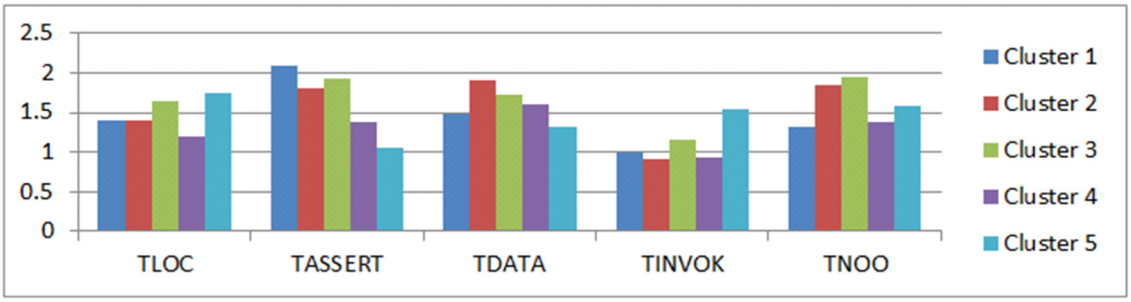


Figure 9: Coefficients de variations (Cv) des métriques de test par cluster, partitionnement selon WMC.

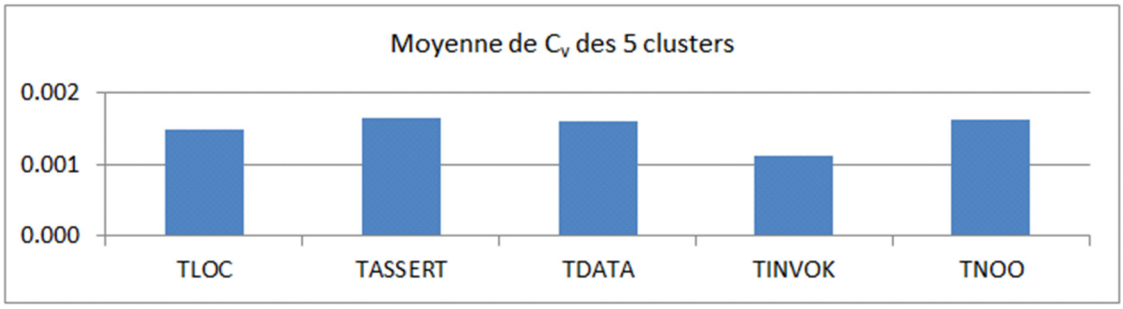


Figure 10: Moyenne des coefficients de variation des métriques de test, partitionnement selon WMC.

6.5.4 Regroupement selon le couplage (CBO) des classes logicielles

Comme dans les deux sous-sections précédentes, nous effectuons un partitionnement univarié (cf. section 3.4.2.1) de l'ensemble des classes logicielles selon leur couplage en 5 clusters. Dans chaque cluster, les classes logicielles ont des valeurs de couplage relativement comparables. Nous les avons ordonnées selon leurs moyennes (de CBO) de la plus faible (cluster 1) à la plus forte (cluster 5). Nous avons aussi calculé l'IVQ de chaque cluster afin de vérifier la représentativité des 6 systèmes logiciels dans les 5 clusters. Les 5 clusters formés par les classes tests dédiées aux classes logicielles sont décrits dans la table 26. Comme pour le partitionnement selon la taille, nous avons calculé les statistiques descriptives des métriques de test, l'indice IVQ ainsi que les coefficients de variation permettant de comparer le degré de volatilité des métriques.

La distribution des observations dans les clusters est différente de celle de TLOC et de WMC. Ici, le cluster 5 compte plus d'observations (20). Les différents indices de variance qualitative sont proches de 1, nous garantissant que les 6 styles sont significativement bien représentés dans les 5 clusters.

Table 26: Analyse de la variance des données de test uniformément partitionnées selon CBO.

		Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
	Nb. Obs.	506	284	153	64	20
	IVQ	0.842	0.946	0.952	0.806	0.762
TLOC	Min	6	9	12	18	16
	Max	1620	2379	2624	1353	4063
	Moy (μ)	91.81	175.62	236.54	149	477.85
	Var (σ^2)	20 999.23	89 260.43	177 940.81	42 464.03	729 461.23
	Coef. of var ($C_v = \sigma/\mu$)	1.59	1.70	1.78	1.38	1.79
TASSERT	Min	0	0	0	1	1
	Max	615	1058	1156	305	391
	Moy (μ)	20.316	46.257	65.523	38.313	93.3
	Var (σ^2)	1798.90	13 538.97	26 170.56	3392.84	10 307.71
	Coef. of var ($C_v = \sigma/\mu$)	2.09	2.52	2.47	1.52	1.09
TDATA	Min	0	0	0	0	0
	Max	482	393	482	223	758
	Moy (μ)	12.844	25.68	33.758	22.047	70.35
	Var (σ^2)	971.85	2788.08	4586.13	925.61	26 077.93
	Coef. of var ($C_v = \sigma/\mu$)	2.427	2.056	2.006	1.38	2.295
TINVOK	Min	0	0	3	1	6
	Max	175	401	399	190	516
	Moy (μ)	25.215	38.757	54.059	35.688	84.05
	Var (σ^2)	685.21	1807.85	3560.74	914.69	11 521.85
	Coef. of var ($C_v = \sigma/\mu$)	1.038	1.097	1.104	0.847	1.277
TNOO	Min	0	1	1	1	1
	Max	164	242	238	54	148
	Moy (μ)	6.78	11.05	19.46	8.66	21.85
	Var (σ^2)	172.84	519.35	1435.99	107.98	1091.43
	Coef. of var ($C_v = \sigma/\mu$)	1.94	2.06	1.95	1.20	1.51

La table 26 (comme les tables 24 et 25 précédentes) montre que les moyennes des différentes métriques croissent avec les clusters (qui sont ordonnés, rappelons-le, selon les valeurs moyennes du couplage des classes logicielles des clusters). Cela montre aussi le lien entre le couplage des classes logicielles et les métriques de test. Dans le cas de ce partitionnement, les métriques TLOC et TINVOK ont les coefficients de variation les plus faibles dans les clusters à l'exception du cluster 4 où TNOO arrive en 2^{ième} position devant TLOC. TASSERT et TDATA sont les métriques qui varient le plus au sein des clusters (figure 11). Le graphe de la figure 12 révèle les moyennes des coefficients de variation (des métriques de test) des clusters et suggère que TINVOK puis TLOC sont ici aussi les moins affectées par le style d'écriture des classes tests.

Figure 11: Coefficients de variations (Cv) des métriques de test par cluster, partitionnement selon CBO.

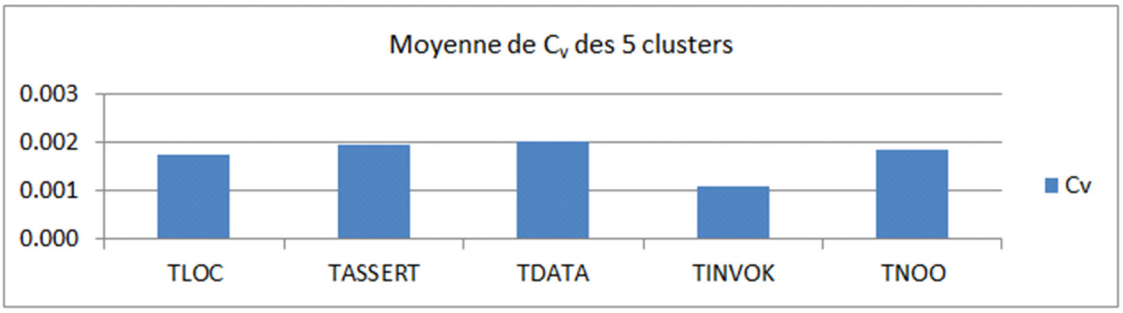


Figure 12: Moyenne des coefficients de variation des métriques de test, partitionnement selon CBO.

L'analyse de la variance selon le regroupement par LOC, WMC et CBO (pris séparément ou en groupe) révèle que les métriques TLOC et TASSERT semblent être les moins fluctuantes vis-à-vis des différents styles d'écriture des tests unitaires adoptés dans les 6 systèmes analysés. Cependant, ce résultat peut aussi être dû au fait que ces métriques sont totalement indépendantes des caractéristiques logicielles considérées dans le sens qu'elles varient peu quel que soit la variation de l'attribut logiciel considéré. Un premier faisceau d'indices tend à rejeter cette hypothèse. Il s'agit de la croissance des métriques monotone de TINVOK et TLOC entre le cluster 1 et le cluster 5, et ceci pour tous les attributs logiciels (LOC, WMC et CBO) considérés. En effet, l'ordonnancement des clusters avait été effectué auparavant, selon la moyenne croissante des valeurs de ces attributs (dans les clusters des classes logicielles). Cela suggère, donc, que les métriques TLOC et TINVOK croissent avec les attributs choisis. Pour confirmer ce résultat, nous allons étudier le lien entre ces attributs logiciels et les métriques de test utilisées dans cette étude. Il s'agit d'un des objectifs de la section suivante.

6.6 Liens entre les métriques logicielles et les métriques de test

Nous analysons dans cette section le lien entre les métriques logicielles et les métriques de test étudiées dans les sections précédentes. L'objectif est double. Il s'agit, d'une part, de vérifier que les métriques de test qui sont ressorties des études (ACP et analyse de variance) précédentes sont bien reliées aux attributs de couplage, de complexité et de taille (et que par conséquent, leur faible variance n'est pas due à leur indépendance vis-à-vis de ces métriques). D'autre part, que l'information indépendante qu'elles capturent (selon l'ACP) est effectivement liée aux attributs logiciels. Nous avons effectué une analyse de corrélations de Spearman et de Pearson pour confirmer ces résultats.

Hypothèse: Il existe un lien significatif entre les métriques de tests unitaires et les 3 attributs logiciels.

Pour vérifier cette hypothèse, nous avons effectué une analyse de corrélation de Pearson que nous avons complétée par une analyse de corrélation de Spearman (décrites dans la section 3.4.1.1). Nous avons déterminé les *valeurs-p* et avons fixé notre seuil de significativité α à 5%. Les corrélations significatives sont indiquées en gras dans la table 27.

Table 27: Corrélations entre métriques logicielles et métriques de test.

	Pearson			Spearman		
	LOC	WMC	CBO	LOC	WMC	CBO
TLOC	**0.449**	**0.459**	**0.18**	**0.429**	**0.439**	**0.321**
ASSERT	**0.312**	**0.343**	**0.149**	**0.393**	**0.42**	**0.236**
TDATA	**0.364**	**0.42**	**0.148**	**0.339**	**0.366**	**0.305**
TINVOK	**0.36**	**0.436**	**0.224**	**0.355**	**0.369**	**0.342**
TNOO	**0.284**	**0.326**	**0.127**	**0.365**	**0.389**	**0.296**

Nous observons dans la table 27 que toutes les corrélations (Spearman ou Pearson) sont (en gras donc) significatives au sens α, leurs *valeurs-p* sont plus grandes que 5%. Les corrélations de Spearman sont légèrement plus élevées en général. Ce constat confirme notre hypothèse à savoir l'existence de lien significatif entre les attributs logiciels et les métriques de test. En regardant les colonnes de la table, nous voyons que la métrique WMC présente des valeurs de corrélation (Pearson et Spearman) relativement plus grandes que les autres métriques, ce qui suggère que la complexité des classes logicielles est fortement liée aux métriques de test. En suivant les lignes de la table, nous remarquons

que les valeurs des corrélations de TLOC sont plus marquées (que celles des autres métriques de test) avec les attributs LOC et WMC et que les corrélations de TINVOK sont légèrement plus élevées (que celles des autres métriques de test) vis-à-vis de l'attribut de couplage de la classe logicielle. Cette remarque, ainsi que le signe positif des corrélations, viennent confirmer les tendances observées à la section précédente entre TLOC, TINVOK et les métriques logicielles. Nous pouvons affirmer de manière raisonnable que les variances relativement faibles de ces métriques observées précédemment ne sont pas dues à l'absence de liens avec les attributs logiciels, mais plutôt (d'après les analyses de variances) à leur volatilité relativement plus faible vis-à-vis des différents styles. Nous avons, au terme de cette troisième analyse, montré que les métriques de test, plus particulièrement TLOC et TINVOK capturent des informations pertinentes, indépendantes et significativement liées aux attributs logiciels.

6.7 Risques pour la validité

Nous avons analysé des données de plus de 1000 classes de test unitaire JUnit dédiées à des classes logicielles. Même si nous pensons que cet échantillon est assez large pour obtenir des résultats significatifs, nous ne prétendons pas que ces résultats peuvent se généraliser à tous les systèmes. Il y a, en effet, un certain nombre de limitations pouvant affecter les résultats obtenus et retreindre leur généralisation.

Le principal risque pour la validité externe a été discuté dans la conclusion de l'étude précédente (section 5.5). Il est en rapport avec la manière d'associer les classes logicielles à leurs classes tests dédiées. D'autre part, le choix de 5 comme nombre de clusters dans l'étude est fait juste pour correspondre aux 5 niveaux d'effort de test (très faible, faible, moyen, élevé, très élevé). En changeant ce nombre, la distribution des classes dans les différents clusters peut changer et affecter nos résultats. Il serait intéressant de reproduire l'étude avec moins de niveaux (exemple: élevé, moyen et faible) ou encore d'utiliser des techniques de classification automatique comme la classification hiérarchique ascendante qui déterminent automatiquement le nombre de clusters.

Les risques pour la validité externe concernent particulièrement le test partiel du logiciel. Rappelons d'abord que toutes les classes ne sont pas testées et qu'ensuite les classes logicielles pour lesquelles des classes de test unitaires JUnit ont été développées ne voient pas toutes leurs méthodes effectivement couvertes par les tests unitaires. En effet, l'analyse de certaines classes logicielles et leurs classes test associées, nous a fait remarquer que les testeurs se concentrent sur les classes larges et complexes (ce qui a été confirmé par les statistiques descriptives) et à l'intérieur de ces classes logicielles, les développeurs ne testent (le plus souvent) que les méthodes complexes. Alors qu'en face, nous utilisons les informations issues de toutes les méthodes des classes logicielles pour déterminer les attributs logiciels (LOC, WMC et CBO).

Le choix de 5 niveaux entraine un risque particulier pour la validité conceptuelle de la démarche. En effet, cette contrainte peut entrainer une variance intra-cluster relativement forte dans les groupes obtenus par les techniques de classification et faire en sorte que les classes logicielles issues du même cluster aient des attributs très différents. Pour réduire l'effet de ce risque, nous pourrions, dans un environnement contrôlé, faire tester un même logiciel par différents groupes de testeurs (adoptant des styles d'écriture différents) et mesurer les variances des métriques de test.

6.8 Conclusion

Dans cette étude, nous avons analysé les classes de tests unitaires JUnit de 6 systèmes logiciels *open source* écrits en JAVA. Nous avons utilisé 5 métriques pour quantifier différentes perspectives reliées à leur code. L'objectif était d'étudier et d'identifier un sous-ensemble de métriques de test indépendantes, offrant des informations pertinentes sur l'effort de test unitaire et qui sont les moins biaisées par les styles qu'adoptent les développeurs lors de l'écriture des classes de test.

Nous avons utilisé l'ACP pour déterminer les métriques les plus indépendantes qui capturent le maximum d'information parmi les 5 métriques de test originales (retenues pour l'expérimentation). Nous avons par la suite analysé la variance de ces métriques de test selon différentes perspectives de taille, de couplage et de complexité des classes

logicielles. Nous avons enfin étudié, grâce à l'analyse de corrélation de Spearman et de Pearson, le lien entre les attributs logiciels et les métriques de test. Il ressort de ces différentes analyses que les métriques TLOC et TINVOK sont les plus indépendantes, capturant le maximum d'information, en étant les moins volatiles vis-à-vis du style des développeurs et les plus liées aux attributs logiciels. Ces résultats ont fait l'objet de publications [56,58].

CHAPITRE 7. PRÉDICTION DES NIVEAUX D'EFFORT DE TESTS UNITAIRES

7.1 Introduction et Objectifs

Nous avons, dans les investigations précédentes, exploré les liens entre les métriques logicielles, incluant les indicateurs de qualité Qi, et la testabilité. La testabilité a été abordée sous l'angle de l'effort d'écriture et de construction des tests unitaires. Les résultats ont montré des liens significatifs entre ces métriques et l'effort d'écriture des tests unitaires. Dans cette section, nous avons affiné l'analyse de ces liens en tentant de prédire les différents niveaux d'effort des tests unitaires. En effet, dans le cadre d'un processus itératif de développement de projet logiciel relativement grand, avec des ressources limitées, la prédiction des niveaux d'effort requis pour tester unitairement les classes logicielles peut apporter une aide aux responsables de projets pour répartir adéquatement les ressources humaines et orienter l'effort de test vers les classes logicielles qui ont en le plus besoin pour assurer la qualité du produit final.

Pour cette étude, nous avons besoin de caractériser les classes tests afin de capturer l'effort qu'elles ont requis pour leur construction et leur écriture. Nous avons utilisé 4 métriques pour quantifier les caractéristiques reliées aux classes de tests unitaires. Deux approches de classification ont été mises en œuvre pour hiérarchiser les classes tests en fonction du niveau de l'effort (d'écriture et de construction) qu'elles requièrent. L'étude empirique menée par la suite fait intervenir 8 systèmes logiciels *open source* écrits en JAVA ayant différents domaines d'applications. Dans ces logiciels, certaines classes ont été unitairement testées grâce au Framework JUnit, et les unités de test sont disponibles. Pour évaluer la capacité des Qi à prédire les différents niveaux de test, nous avons utilisé la régression logistique univariée binaire, la régression logistique univariée multinomiale et la régression linéaire univariée. Les performances des Qi ont été comparées à celles des métriques OO issues de la suite CK.

7.2 Prédiction des niveaux d'effort de tests unitaires

Les métriques logicielles peuvent être utilisées pour prédire la testabilité unitaire des classes [101,102,138]. Elles mesurent, en effet, certains attributs logiciels pouvant aider à estimer l'effort de test. Un moyen efficace d'y parvenir est de construire des modèles qui peuvent identifier les parties critiques du logiciel, nécessitant un effort de test relativement important [138]. Différentes recherches ont été menées [56,101-103,128,136] et ont montré le lien significatif de l'effort de test avec les métriques logicielles. Cependant, aucune étude n'a, à notre connaissance, affiné les investigations pour prédire les différents niveaux de l'effort de test que requièrent les classes logicielles. La prédiction des niveaux d'effort de test peut, en effet, servir d'outil d'aide à la décision pour les responsables de projets soucieux de la bonne qualité de leurs produits logiciels. Elle peut être d'une aide particulièrement intéressante dans la planification, le suivi des activités de test, l'allocation de ressources et dans la détection des classes critiques du logiciel sur lesquelles les développeurs doivent apporter des soins particuliers pour assurer la qualité du logiciel. La testabilité étant une notion complexe, nous l'abordons ici sous la perspective de la construction des tests unitaires, plus particulièrement sous l'angle des différents niveaux d'effort lié à l'écriture des classes tests JUnit.

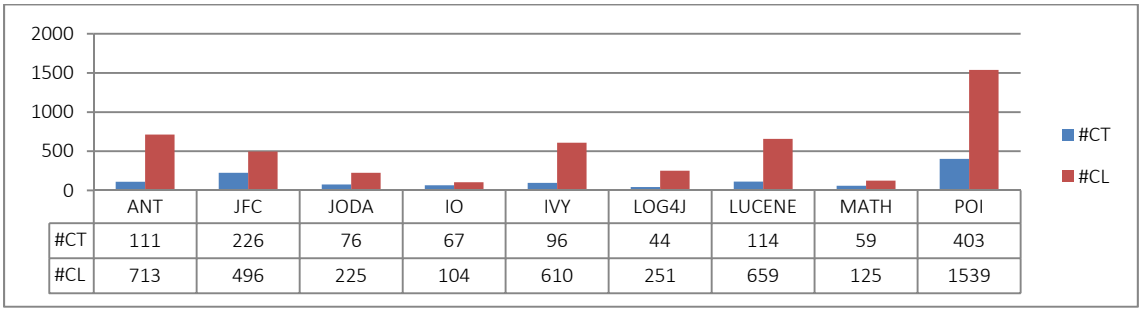
7.3 Métriques, systèmes logiciels et statistiques descriptives

Pour notre expérimentation, nous avons choisi de comparer les performances des indicateurs de qualité (Qi) à 6 métriques logicielles OO issues de la suite de CK incluant la taille en termes de lignes de code (LOC). Les métriques choisies se composent de 2 métriques liées au couplage (CBO et RFC), d'une métrique de cohésion (LCOM), d'une métrique de complexité (WMC), de 2 métriques d'héritages (DIT et NOC) et enfin de la métrique de taille LOC qui servira de base de comparaison. Ces 6 métriques ainsi que LOC connaissent un engouement important de la part de la communauté des chercheurs et des développeurs. Elles sont intégrées de plus en plus aux environnements de développement.

Face aux 8 (incluant les Qi) métriques logicielles, 4 métriques de test ont été choisies pour mesurer différents aspects de l'effort d'écriture et de construction des classes tests.

Nous surveillerons particulièrement les résultats de TLOC et TINVOK dont les investigations de la section précédente ont montré la pertinence, l'indépendance et le peu de volatilité de l'information qu'elles véhiculent. Les métriques de test et les caractéristiques qu'elles capturent ont été décrites dans la section 3.2.2.

La démarche empirique de la section suivante sera menée sur un ensemble de 8 systèmes logiciels *open source* écrits en JAVA, parmi ceux décrits dans la section 3.1. Nous avons apparié les classes tests aux classes logicielles auxquelles elles étaient dédiées par le mécanisme décrit en 3.3. La figure 13 fournit un aperçu de la taille des systèmes (#CL) et celle de leurs suites de tests unitaires (#CT) associées en termes de nombre de classes.



	ANT	JFC	JODA	IO	IVY	LOG4J	LUCENE	MATH	POI
#CT	111	226	76	67	96	44	114	59	403
#CL	713	496	225	104	610	251	659	125	1539

Figure 13: Distribution des lignes des systèmes et des classes de test.

Les systèmes MATH et IO ajoutés à l'échantillon précèdent sont relativement petits. POI reste le plus grand système et IO est le plus petit en termes de nombre de classes. Seule une partie des classes logicielles est testée dans chaque logiciel.

Les tables 28 et 29 contiennent les statistiques descriptives des 4471 classes logicielles étudiées ainsi que celles du sous-ensemble formé des classes logicielles pour lesquelles des classes de test unitaire JUnit ont été développées (1152). Nous avons exclu de ces ensembles les observations (classes) pour lesquelles LCOM n'a pu être calculée à cause de sa définition. Les moyennes globales des métriques des classes logicielles explicitement testées sont plus élevées que celles de toutes les classes, sauf pour la métrique NOC. Ce constat suggère que les métriques de test et la métrique NOC seront faibles.

Table 28: Statistiques descriptives des métriques de l'ensemble des classes logicielles.

	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
Nb. Obs.	4471	4471	4471	4471	4471	4471	4471
Min	0	0	0	0	0	0	1

Max	168	131 982	557	679	7	238	3560
Moy (μ)	6.948	141.149	18.361	54.352	1.933	1.500	83.647
Écart-Type (σ)	9.851	2620.868	31.594	71.586	1.332	8.246	170.563

Table 29: Statistiques descriptives des métriques des classes logicielles explicitement testées.

	CBO	**LCOM**	**WMC**	**RFC**	**DIT**	**NOC**	**LOC**
Nb. Obs.	1152	1152	1152	1152	1152	1152	1152
Min	0	0	0	5	0	0	2
Max	111	13 524	557	677	6	57	2644
Moy (μ)	11.148	227.791	32.852	83.302	2.209	0.846	166.637
Écart-Type (σ)	13.147	1030.981	45.278	94.893	1.277	4.348	232.387

7.4 Démarches empiriques

7.4.1 Analyse de corrélations

Pour explorer les relations entre les métriques de test m_t et les métriques logicielles m_s, incluant Qi et LOC, nous avons effectué l'analyse de corrélation de Pearson que nous avons complétée avec celle de Spearman. Les deux méthodes d'analyse ainsi que leurs méthodes de calcul sont présentées dans la section 3.4.1.1. Nous voulons tester l'hypothèse suivante:

Hypothèse: Il existe une relation entre les métriques de code source (m_s) et les métriques de test (m_t).

Les tables 30 et 31 présentent respectivement les résultats des corrélations de Pearson et de Spearman entre les métriques de test et les métriques logicielles. Nous obtenons un ensemble de 32 valeurs liant les paires $<m_s, m_t>$ pour chaque analyse. Le seuil de significativité α est fixé à 5% et les corrélations significatives (*valeur-p* > 5%) sont en gras. En analysant les résultats des deux tables, nous pouvons voir 3 groupes de métriques logicielles selon la force de leurs liens de corrélation.

- Le premier ensemble est formé des métriques LOC, Qi et WMC: ces métriques ont des corrélations de Spearman ou Pearson significatives pour tous les systèmes selon leurs *p-values*. Le signe négatif des corrélations de Qi indique que cette métrique à des variations inverses de celles des métriques de test. Ce qui est logique, de par leur formulation, une valeur des indicateurs de qualité proche de 1 suggère une classe de qualité (au sens test) relativement bonne alors qu'un Qi proche de 0 suggère des classes relativement complexes

et couplées qui nécessitent plus d'effort de test. Notons que les corrélations des 3 métriques de cet ensemble sont comparables, en général, dans la table 31. LOC donne de meilleures corrélations en considérant les corrélations de Pearson (table 30). Ces dernières sont, en effet, plus sensibles à l'effet d'échelle. En éliminant ce biais avec la corrélation de Spearman, nous constatons que dans la plupart des systèmes les indicateurs de qualité donnent des corrélations légèrement meilleures ou comparables à celles de LOC, et que les plus fortes valeurs de corrélations sont observées avec les métriques TLOC et TINVOK, avec un maximum de 0.838 entre WMC et TLOC du système IO dans la table 31 et de 0.936 pour la table 30 entre LOC et TLOC de IO.

- Le deuxième sous-ensemble de métriques est formé de CBO, RFC et LCOM. Il regroupe les métriques dont les corrélations avec les métriques de test sont significatives pour presque tous les systèmes considérés. Les métriques CBO et RFC capturent le couplage, même si RFC capture la dimension récursive des liens entre classes (non capturée par CBO). Dans la plupart des résultats de la table 31, nous constatons que les liens entre ces deux métriques et les métriques de test suivent les mêmes tendances. Ils sont significatifs ou non en même temps face aux métriques de test. Pour tous les systèmes, ces deux métriques sont significativement liées aux métriques de test TLOC et TINVOK. Les métriques de cet ensemble présentent plus de corrélations significatives avec les métriques de test que des corrélations non significatives.

Table 30: Corrélations de Pearson entre les métriques de test et les métriques logicielles par système.

		Corrélations de Pearson							
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	TASSERT	**-0.416**	**0.231**	**0.239**	**0.404**	0.086	-0.163	-0.007	**0.389**
	TDATA	**-0.384**	**0.264**	0.152	**0.357**	0.090	-0.061	0.001	**0.325**
	TLOC	**-0.419**	**0.352**	**0.212**	**0.423**	0.184	0.007	0.026	**0.432**
	TINVOK	**-0.244**	**0.295**	0.024	**0.232**	-0.024	**0.311**	-0.111	**0.298**
JFC	TASSERT	**-0.485**	**0.315**	**0.613**	**0.635**	0.096	-0.023	**0.256**	**0.595**
	TDATA	**-0.267**	0.088	**0.334**	**0.341**	-0.061	-0.095	0.054	**0.304**
	TLOC	**-0.472**	**0.294**	**0.567**	**0.605**	0.110	-0.005	**0.162**	**0.579**
	TINVOK	**-0.556**	**0.477**	**0.717**	**0.747**	**0.257**	**0.161**	**0.192**	**0.703**
JODA	TASSERT	**-0.670**	**0.471**	**0.805**	**0.710**	**0.597**	**0.331**	-0.141	**0.419**
	TDATA	**-0.595**	**0.392**	**0.736**	**0.667**	**0.470**	**0.235**	-0.085	**0.386**
	TLOC	**-0.716**	**0.527**	**0.813**	**0.769**	**0.575**	**0.273**	-0.091	**0.453**
	TINVOK	**-0.654**	**0.565**	**0.792**	**0.675**	**0.676**	**0.417**	-0.103	**0.366**
IO	TASSERT	**-0.543**	**0.359**	**0.464**	**0.718**	**0.508**	-0.238	-0.020	**0.667**
	TDATA	**-0.526**	**0.827**	**0.876**	**0.843**	**0.807**	-0.218	0.032	**0.857**
	TLOC	**-0.732**	**0.831**	**0.816**	**0.925**	**0.826**	**-0.322**	0.157	**0.936**
	TINVOK	**-0.688**	**0.877**	**0.841**	**0.872**	**0.848**	**-0.280**	0.183	**0.900**
IVY	TASSERT	-0.164	0.106	-0.026	0.092	**0.326**	0.044	-0.024	**0.446**
	TDATA	**-0.318**	0.102	-0.014	0.192	**0.351**	**0.245**	0.018	**0.222**
	TLOC	**-0.308**	0.133	-0.029	0.153	**0.412**	0.197	-0.011	**0.401**
	TINVOK	**-0.369**	**0.230**	-0.033	**0.207**	**0.547**	**0.369**	0.016	**0.335**
LUCENE	TASSERT	**-0.335**	**0.531**	**0.735**	**0.744**	**0.550**	-0.125	0.039	**0.655**
	TDATA	-0.166	**0.454**	**0.915**	**0.786**	**0.627**	-0.051	0.043	**0.643**
	TLOC	**-0.455**	**0.456**	**0.914**	**0.802**	**0.618**	-0.056	0.061	**0.809**
	TINVOK	**-0.196**	**0.539**	**0.929**	**0.843**	**0.675**	-0.119	0.074	**0.701**
MATH	TASSERT	**-0.681**	0.198	**0.549**	**0.589**	**0.470**	**-0.440**	-0.107	**0.561**
	TDATA	**-0.637**	0.000	**0.367**	**0.792**	**0.550**	-0.240	-0.081	**0.800**
	TLOC	**-0.782**	**0.319**	**0.592**	**0.826**	**0.657**	**-0.479**	-0.106	**0.806**
	TINVOK	**-0.723**	**0.357**	**0.582**	**0.780**	**0.744**	**-0.483**	-0.149	**0.766**
POI	TASSERT	**-0.525**	**0.393**	**0.548**	**0.625**	**0.517**	**-0.151**	-0.001	**0.572**
	TDATA	**-0.278**	**0.203**	**0.172**	**0.388**	**0.235**	-0.101	-0.039	**0.455**
	TLOC	**-0.352**	**0.252**	**0.280**	**0.493**	**0.304**	**-0.191**	-0.021	**0.535**
	TINVOK	**-0.632**	**0.526**	**0.498**	**0.648**	**0.634**	-0.094	-0.047	**0.534**

- Le troisième et dernier ensemble est formé des métriques d'héritage DIT et NOC. Tandis que NOC ne présente qu'une seule corrélation de Spearman significative, DIT présente quant à elle quelques valeurs intéressantes avec les métriques de test, particulièrement la métrique TINVOK. Cependant, le signe changeant de cette corrélation significative reste difficile à interpréter. Il pourrait être lié au fait que l'utilisation de l'héritage est une bonne pratique en programmation orientée objet, mais un abus crée des couplages très forts entre les classes d'une même hiérarchie ce qui rend leur test complexe et réduit leur qualité.

Table 31: Corrélations de Spearman entre les métriques de test et les métriques logicielles par système.

		Corrélations de Spearman							
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	TASSERT	-0.391	0.135	0.387	0.391	0.071	-0.203	0.034	0.391
	TDATA	-0.364	0.176	0.302	0.358	0.109	-0.153	0.066	0.340
	TLOC	-0.554	0.394	0.497	0.566	0.342	0.006	0.048	0.582
	TINVOK	-0.267	0.384	0.154	0.292	0.434	0.421	-0.057	0.349
JFC	TASSERT	-0.443	0.261	0.415	0.453	0.197	0.069	0.224	0.414
	TDATA	-0.288	0.105	0.216	0.280	0.030	-0.018	-0.039	0.235
	TLOC	-0.452	0.305	0.364	0.450	0.257	0.166	0.106	0.437
	TINVOK	-0.487	0.376	0.439	0.479	0.352	0.263	0.090	0.444
JODA	TASSERT	-0.744	0.588	0.767	0.737	0.736	0.415	-0.195	0.726
	TDATA	-0.648	0.579	0.670	0.653	0.593	0.319	-0.132	0.606
	TLOC	-0.800	0.630	0.789	0.808	0.733	0.359	-0.099	0.764
	TINVOK	-0.700	0.643	0.757	0.680	0.770	0.522	-0.067	0.635
IO	TASSERT	-0.634	0.416	0.493	0.635	0.518	-0.288	0.043	0.654
	TDATA	-0.668	0.700	0.473	0.671	0.574	-0.266	-0.024	0.696
	TLOC	-0.813	0.658	0.604	0.838	0.640	-0.395	0.124	0.827
	TINVOK	-0.743	0.644	0.629	0.759	0.727	-0.264	0.073	0.769
IVY	TASSERT	-0.376	0.167	0.217	0.379	0.345	0.159	0.015	0.389
	TDATA	-0.302	0.298	0.141	0.337	0.446	0.262	0.072	0.368
	TLOC	-0.366	0.323	0.253	0.354	0.445	0.306	0.092	0.382
	TINVOK	-0.380	0.476	0.281	0.384	0.630	0.409	0.001	0.404
LUCENE	TASSERT	-0.560	0.239	0.276	0.492	0.436	0.039	0.031	0.495
	TDATA	-0.287	0.308	0.229	0.295	0.413	0.209	0.048	0.292
	TLOC	-0.464	0.242	0.274	0.413	0.422	0.105	0.123	0.470
	TINVOK	-0.409	0.402	0.324	0.417	0.505	0.068	0.177	0.445
MATH	TASSERT	-0.650	0.481	0.311	0.674	0.484	-0.606	-0.192	0.657
	TDATA	-0.295	0.030	0.003	0.320	0.206	-0.156	-0.169	0.281
	TLOC	-0.711	0.531	0.310	0.714	0.459	-0.675	-0.165	0.704
	TINVOK	-0.599	0.473	0.308	0.628	0.563	-0.495	-0.253	0.606
POI	TASSERT	-0.464	0.280	0.140	0.400	0.365	-0.100	-0.005	0.397
	TDATA	-0.264	0.393	-0.052	0.284	0.219	-0.133	0.023	0.268
	TLOC	-0.360	0.392	-0.004	0.398	0.237	-0.327	0.017	0.397
	TINVOK	-0.528	0.445	0.220	0.480	0.537	-0.004	-0.036	0.485

Suivant les systèmes (dans la table 31), nous remarquons que les classes tests de JODA présentent les plus fortes valeurs de corrélations, suivies de celles d'IO et de MATH. Rappelons que JODA compte en moyenne plus de 2 lignes de test pour une ligne de code testée (Table 14). Ce qui pourrait être un bon signe du soin apporté aux classes de test. Cette table montre aussi que les métriques des classes tests de JODA sont significativement liées à presque toutes les métriques logicielles sauf NOC. Les plus faibles corrélations sont observées pour le système IVY. Toujours suivant les lignes de la table 31, mais niveau métrique de test, on remarque que TDATA présente les plus faibles valeurs de corrélation, en général, si toutes fois elles sont significatives, alors que TLOC et TINVOK offrent les plus fortes valeurs de corrélations. Ce résultat est vrai même pour IO et MATH (dont on n'a pas

étudié les métriques JUnit dans la section précédente). Ce qui conforte les résultats de la section précédente.

7.4.2 Hiérarchisation des classes tests

Nous avons mis en œuvre deux approches de catégorisations des classes tests afin de déterminer le niveau d'effort de test unitaire requis pour chaque classe. Nous avons adopté une approche basée sur la moyenne (MEAN) et une autre basée sur la classification automatique (KMEAN). Ces techniques nous permettent au final d'obtenir 5 catégories de classes que nous désignerons par les niveaux d'effort très faible (*TF*), faible (*F*), moyen (*M*), élevé (*E*) et très élevé (*TE*). Les approches utilisent les 4 métriques de test pour construire ces niveaux. Nous voulons explorer la capacité des métriques logicielles (Qi y compris) à prédire ces différents niveaux.

7.4.2.1 L'approche basée sur la moyenne (MEAN)

Les 5 niveaux de l'approche MEAN sont construits comme suit pour chaque système:

- Catégorie 5: Cette catégorie regroupe l'ensemble des classes logicielles dont les classes test correspondantes satisfont les 4 conditions suivantes:
  (1) TLOC ≥ moyenne de la variable TLOC du système,
  (2) TASSERT ≥ moyenne de la variable TASSERT du système,
  (3) TDATA ≥ moyenne de la variable TDATA du système, et
  (4) TINVOK ≥ moyenne de la variable TINVOK du système.

  Cette catégorie correspondra au niveau d'effort *TE*.
- Catégorie 4: seules 3 des 4 conditions précédentes sont vérifiées. Ce qui correspond au niveau d'effort *E*.
- Catégorie 3: seules 2 des 4 conditions précédentes sont vérifiées. Ce qui correspond au niveau d'effort *M*.
- Catégorie 2: seule 1 des 4 conditions précédentes est vérifiée. Ce qui correspond au niveau d'effort *F*.
- Catégorie 1: aucune des 4 conditions précédentes n'est vérifiée. Ce qui correspond au

niveau d'effort *TF*.

La table 32 suivante donne la répartition des classes logicielles sur ces différents niveaux.

Table 32: Distribution des classes logicielles selon MEAN.

	Categ. 1 (TF)	Categ. 2 (F)	Categ. 3 (M)	Categ. 4 (E)	Categ. 5 (TE)
ANT	20 (18.02%)	50 (45.05%)	15 (13.51%)	15 (13.51%)	11 (9.91%)
JFC	119 (52.65%)	28 (12.39%)	23 (10.18%)	25 (11.06%)	31 (13.72%)
JODA	45 (59.21%)	3 (3.95%)	11 (14.47%)	2 (2.63%)	15 (19.74%)
IO	33 (49.25%)	12 (17.91%)	6 (8.96%)	6 (8.96%)	10 (14.93%)
IVY	53 (55.79%)	14 (14.74%)	9 (9.47%)	6 (6.32%)	13 (13.68%)
LUCENE	59 (51.75%)	21 (18.42%)	12 (10.53%)	14 (12.28%)	8 (7.02%)
MATH	31 (52.54%)	6 (10.17%)	6 (10.17%)	9 (15.25%)	7 (11.86%)
POI	165 (47.01%)	66 (18.80%)	41 (11.68%)	40 (11.40%)	39 (11.11%)

Nous remarquons que la répartition des classes est très différente d'un système à l'autre et que les catégories contiennent des nombres d'observations très variables. Cependant, une tendance se dessine. Les premières catégories contiennent généralement plus d'observations que les dernières. En d'autres termes, il y a plus de classes requérant un effort de test relativement faible que de classes requérant un niveau relativement très élevé d'effort de test. Il existe, par ailleurs, des similarités de répartitions de classes selon les catégories pour certains systèmes. La catégorie 1 (TF) contient au tour de 50% des classes logicielles à l'exception d'ANT qui en compte seulement 18%. La catégorie 2 d'ANT voit se regrouper 45% de ses classes logicielles contre environ 18% pour POI, LUCENE et IO. JODA fait figure d'exception avec seulement 3.95%. Les pourcentages de la catégorie 3 (M) tournent autour de 10%. La catégorie 4 voit aussi JODA faire figure d'exception avec un faible taux: 2.63%, alors que pour les autres systèmes, nous avons une répartition autour de 10% des observations (à 5% près) dans cette catégorie. Dans la catégorie 5 (TE), JODA fait aussi figure d'exception encore et compte près de 20% d'observations, soit la plus forte valeur. Les résultats si marqués de JODA peuvent s'expliquer par les corrélations entre ces métriques de test observées lors de l'analyse ACP de la section précédente. Une forte corrélation (positive) entre les métriques de test, entraine une synchronisation de leurs variations et de ce fait même la vérification des 4 conditions constitutives des 5 catégories. Comme résultat, la plupart des classes de JODA se retrouvent dans les catégories à l'extrême (TE, TF) tandis que peu d'observations se retrouvent dans les catégories intermédiaires.

Dans la figure 14 suivante, nous avons ramené les moyennes des valeurs (des métriques logicielles) de chaque catégorie à la même échelle (en divisant par le maximum) afin d'observer les tendances des métriques (Qi compris) des classes logicielles dans les 5 différents niveaux.

Nous constatons que les moyennes de taille (LOC), de complexité (WMC) et des Qi (des différents niveaux) sont croissantes monotones de la catégorie 1 à la catégorie 5 pour tous les systèmes sauf LUCENE. Ce résultat suggère que les classes logicielles des catégories 5 sont les plus grandes et les plus complexes en termes de LOC et de WMC. Ces classes requièrent un effort de test relativement élevé. Les métriques logicielles restantes ont des patrons de variation très irréguliers par rapport à l'ordre des catégories. La moyenne du couplage (CBO) des catégories 1 à 5 croît pour les 3 systèmes JFC, JODA et IO uniquement. Le patron le plus irrégulier revient à la métrique NOC, ses variations semblent totalement aléatoires et indépendantes de l'ordonnancement des catégories. Du point de vue des systèmes, les patrons les plus réguliers (croissance ou décroissance monotone) sont observés pour IO. Pour ce système, les classes logicielles avec les niveaux d'effort les plus élevés présentent de faibles valeurs moyennes de Qi et DIT. Ces classes sont aussi relativement larges, complexes, fortement couplées selon (LOC, WMC, RFC et CBO) et aussi peu cohésives dans le sens LCOM.

Finalement, les variations monotones ou irrégulières des graphiques de la figure 14 sont à mettre en rapport avec (entre autres) les corrélations entre métriques logicielles et métriques de test. En effet, l'analyse des tables 30 et 31 montrait de fortes corrélations entre Qi, WMC, LOC et les métriques de test TLOC, TASSERT, TDATA et TINVOK (qui sont à la base de cette catégorisation), alors que NOC ne présentait presque aucun lien significatif avec ces mêmes métriques.

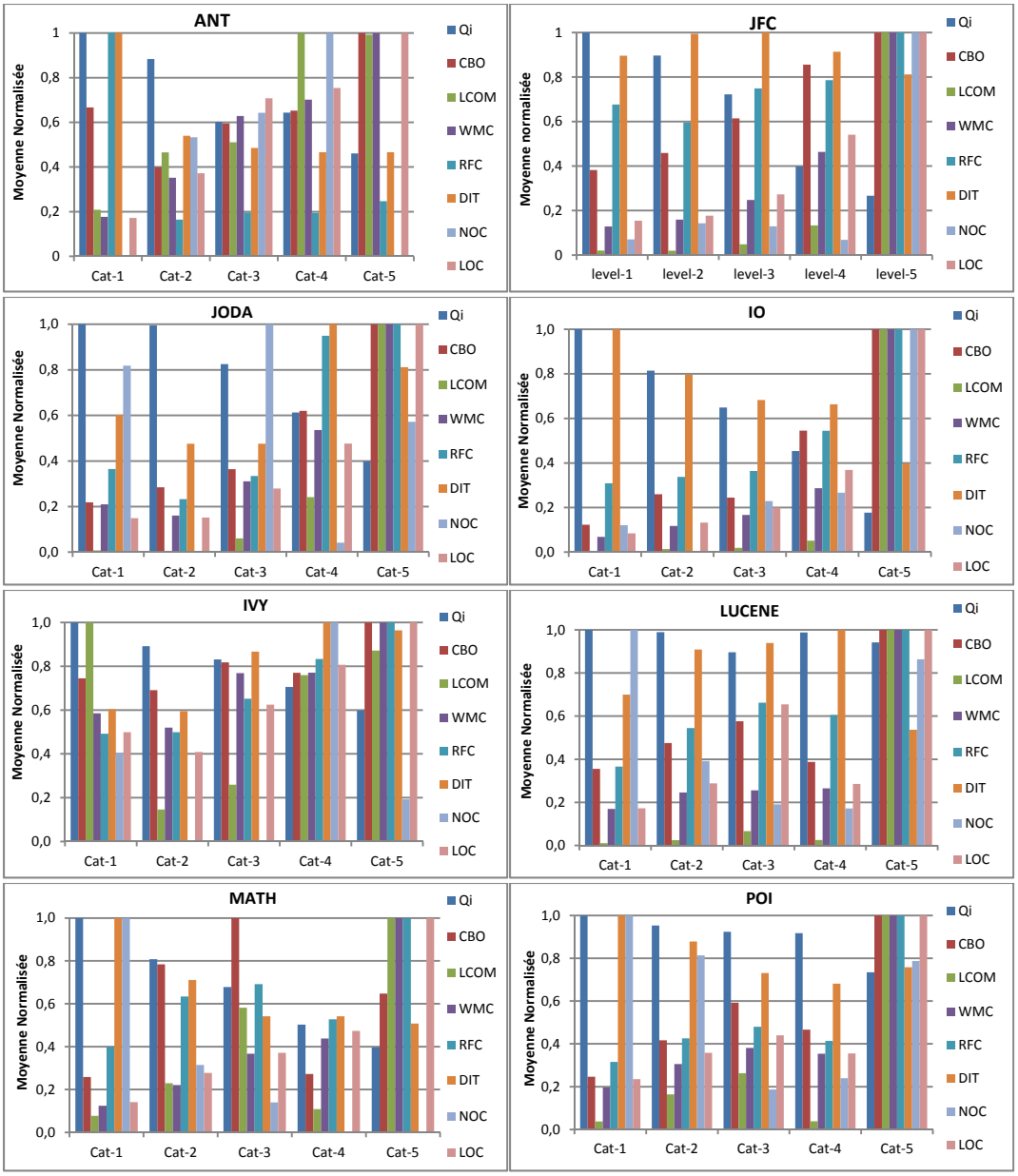


Figure 14: Distribution de la moyenne des métriques selon les niveaux basés sur la moyenne MEAN.

7.4.2.2 L'approche basée sur le regroupement K-Mean (KMEAN)

Cette technique de regroupement, présentée dans la section 3.4.2.1, est utilisée ici sur les données de métriques de test des systèmes. Le nombre de clusters a été fixé à 5 pour correspondre aux 5 niveaux cités dans la section précédente. Les observations (classes tests) sont réparties en 5 groupes dans lesquels chaque classe appartient au cluster dont la moyenne est la plus proche. Il s'agit d'une manière naturelle de regrouper les observations en différents clusters. Au final, les classes du même groupe sont plus proches entre elles qu'avec celles d'autres groupes. Nous voulons vérifier si les propriétés des clusters (niveaux) obtenus reflètent bien le niveau de complexité des classes logicielles correspondantes. Nous pouvons, en effet, intuitivement espérer que les tests unitaires correspondants aux classes complexes soient relativement complexes et les tests unitaires correspondants aux classes logicielles simples soient relativement simples.

Après le regroupement KMEAN, nous avons analysé les différents clusters pour les ordonner, selon les moyennes des métriques tests, en différents niveaux (catégories) d'effort de test (construction et écriture), du plus faible 1 (TF) au plus fort 5 (TE). En observant la figure 15, on constate que la relation d'ordre n'est pas totale au sein des différents clusters (mais plutôt satisfaisante) dans le sens que du niveau 1 au niveau 5, toutes les moyennes des métriques de test ne sont pas croissantes monotones. Nous avons pu résoudre les quelques conflits observés en ordonnançant en priorité selon les (moyennes des) métriques TLOC puis TINVOK. Nous justifions ce choix par l'indépendance, le faible biais et la pertinence de l'information que véhiculent ces métriques ; il s'agit de propriétés constatées lors de l'étude précédente sur les métriques de test.

La figure 15 montre, en effet, les variations des moyennes, des métriques de test, pour chacun des systèmes, et ceci dans les 5 clusters des classes tests qui ont permis d'ordonnancer les classes logicielles correspondantes en niveaux d'effort requis. Nous avons eu 3 cas de conflits de classification:

- La métrique TASSERT pour IO: Le conflit concerne le passage du cluster 3 au cluster 4. La moyenne de la métrique TASSERT décroit légèrement. En priorisant l'ordonnancement de TLOC, nous avons assigné aux clusters 3 et 4 les niveaux de moyen (M) et fort (F). Ce choix correspond aussi à la tendance de la majorité des métriques de test, qui croissent effectivement dans cet ordre entre le cluster 3 et le cluster 4.

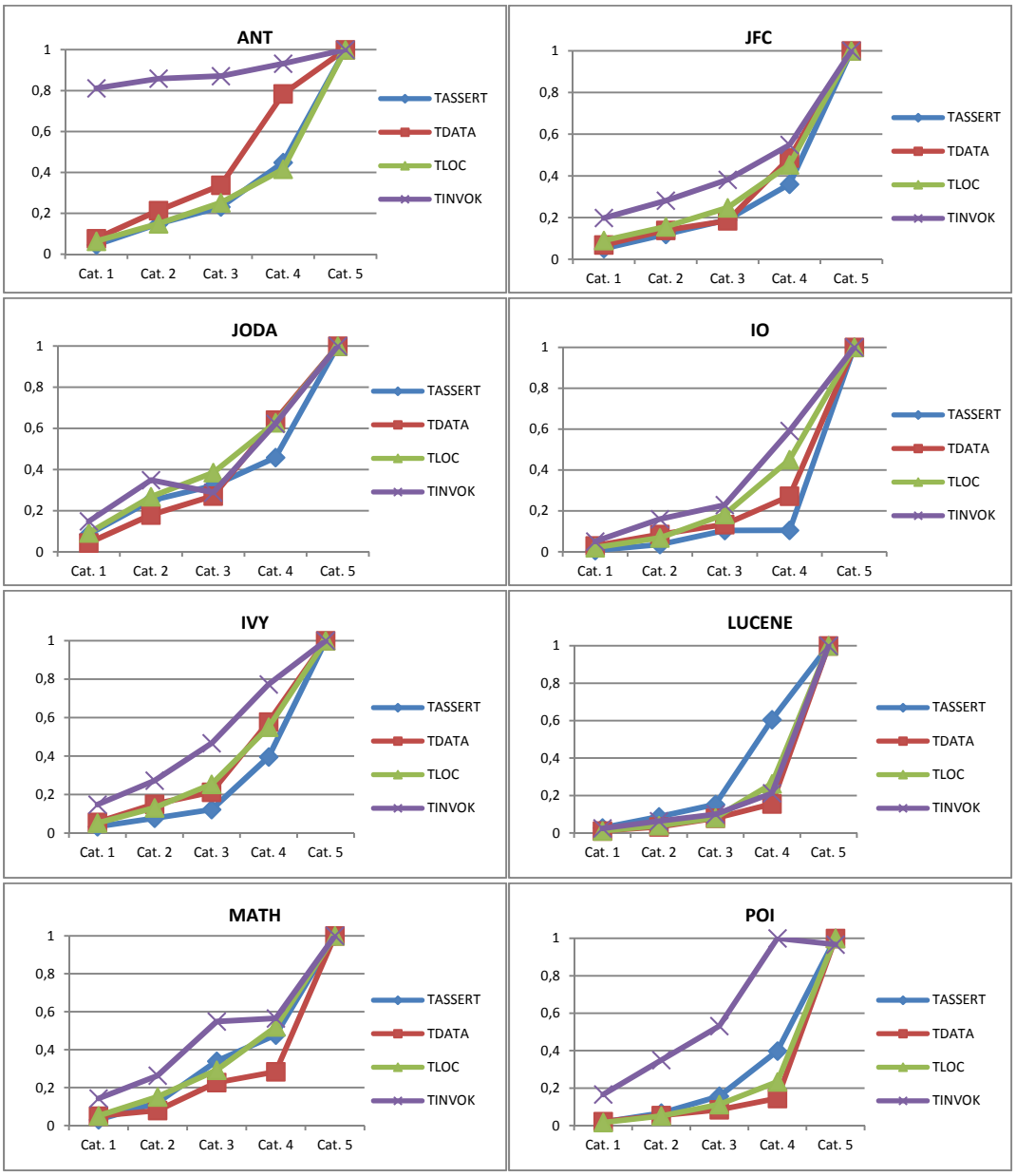


Figure 15: Moyennes des métriques tests selon les clusters KMEAN.

- La métrique TINVOK pour le système JODA: Le conflit concerne le passage de la catégorie 2 à la catégorie 3. La moyenne de TINVOK fléchit. Dans ce cas aussi, nous avons ordonnancé selon TLOC, et l'ordre obtenu correspond à la tendance de la majorité des métriques de test.
- La métrique TINVOK pour le système POI: Le conflit concerne le passage de la catégorie 4 à la catégorie 5 ou la moyenne de TINVOK fléchit de nouveau. En marquant la catégorie 5 comme TE (très élevé), nous avons abondé dans le sens des variations croissantes de la majorité des métriques.

TINVOK présente une certaine indépendance vis-à-vis des autres métriques comme l'a montré l'ACP de l'étude précédente, ce qui peut expliquer, entre autres, pourquoi cette métrique ne suit pas toujours les tendances monotones de la majorité des métriques.

Nous avons finalement ordonnancé de manière satisfaisante les 5 clusters de classes tests de TF à TE selon l'effort d'écriture et de construction qu'elles ont nécessité. La table 33 donne la répartition des classes logicielles correspondantes sur les différentes catégories ordonnées en niveaux d'efforts de TF à TE. La table fournit le nombre et le pourcentage des classes (par rapport au système) de chaque catégorie, pour les 8 systèmes.

Table 33: Distribution des classes logicielles selon KMEAN.

	Categ. 1 (TF)	Categ. 2 (F)	Categ. 3 (M)	Categ. 4 (E)	Categ. 5 (TE)
ANT	49 (44.14%)	30 (27.03%)	17 (15.32%)	10 (9.01%)	5 (4.50%)
JFC	87 (38.50%)	80 (35.40%)	41 (18.14%)	12 (5.31%)	6 (2.65%)
JODA	42 (55.26%)	11 (14.47%)	7 (9.21%)	6 (7.89%)	10 (13.16%)
IO	25 (37.31%)	21 (31.34%)	15 (22.39%)	4 (5.97%)	2 (2.99%)
IVY	49 (51.58%)	21 (22.11%)	15 (15.79%)	6 (6.32%)	4 (4.21)
LUCENE	54 (47.37%)	41 (35.96%)	14 (12.28%)	4 (3.51%)	1 (0.88%)
MATH	28 (47.46%)	12 (20.34%)	9 (15.25%)	6 (10.17%)	4 (6.78%)
POI	220 (62.68%)	91 (25.93%)	29 (8.26%)	10 (2.85%)	1 (0.28%)

Comme pour la distribution des classes logicielles de la table 32, on peut constater que la distribution des 5 catégories diffère d'un système à l'autre. Il y a cependant des similarités de distribution entre certains systèmes. La première catégorie (niveau TF) contient les plus grands nombres de classes avec le nombre de classes logicielles qui varie de 37.31% (pour IO) à 62.68% (pour POI) des classes de chaque système. Ce taux diminue

régulièrement (de la catégorie 1 à 5) pour presque tous les systèmes sauf JODA dont la catégorie 5 (13.16%) contient plus d'observations que les catégories 3 et 4 (resp. 9.21% et 7.89%). Nous avions eu la même observation dans la table 32. Ce constat pourrait s'expliquer aussi par les corrélations significatives des métriques de test de JODA entre elles. Les distributions selon MEAN et KMEAN des tables 32 et 33 diffèrent. La catégorie 5 de la classification KMEAN contient beaucoup moins de classes logicielles (33 au total) que celles de la catégorie 5 de MEAN (134 classes logicielles au total). Par rapport aux systèmes, nous constatons qu'ANT compte beaucoup moins de classes dans la catégorie 2 de la classification selon KMEAN (27.03% contre 45.05% pour MEAN). JFC compte aussi moins de classes dans la catégorie 1 et beaucoup plus dans la catégorie 2.

Pour étudier les caractéristiques des classes logicielles des différentes catégories, nous avons dans les graphes de la figure 16 représenté l'évolution de la moyenne (normalisée à 1) par catégorie de chaque métrique logicielle et pour chacun des 8 systèmes. Nous observons des résultats similaires à la classification basée sur la moyenne de la figure 14. La taille et la complexité des classes croissent de manière monotone suivant l'ordre des catégories, tandis que les valeurs de Qi décroissent de manière monotone dans cet ordre. Les patrons de variations des métriques logicielles de LUCENE deviennent réguliers, contrairement aux observations faites sur ce même système dans la figure 14.

La métrique NOC montre, ici aussi, le patron de variation le plus irrégulier (en dents de scie). Ce résultat n'est guère surprenant au vu des faibles corrélations (peu significatives) de cette métrique vis-à-vis des métriques tests à l'origine de cette classification. JFC et LUCENE ont les patrons de variations les plus monotones selon cette classification.

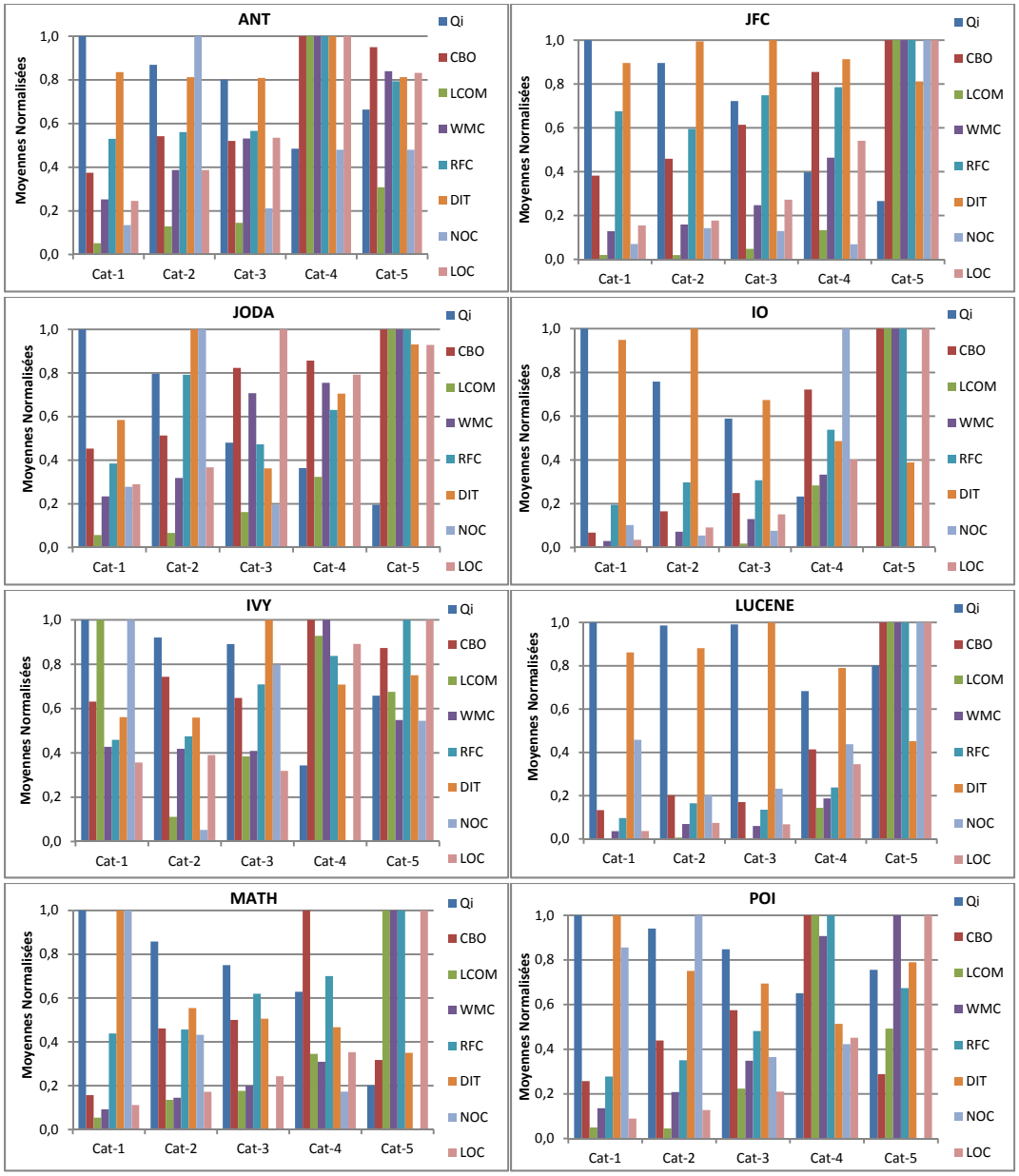


Figure 16: Distribution de la moyenne des métriques selon les niveaux basés sur la moyenne KMEAN.

7.5 Effet des métriques logicielles sur l'effort de test unitaire

Nous présentons, dans cette section, l'étude empirique que nous avons menée pour évaluer les effets individuels des métriques de code source (incluant les Qi) sur la testabilité unitaire des classes logicielles en termes de niveaux d'efforts (d'écriture et de construction des tests unitaires). Nous avons utilisé 3 méthodes de régression univariée parmi celles décrites dans la section 3.4.1. Il s'agit de la régression logistique univariée binaire, de la régression linéaire, et de la régression logistique univariée multinomiale. Ces méthodes ont été appliquées aux données pour déterminer les effets individuels de chacune des métriques logicielles pour identifier celles qui sont significativement reliées aux différents niveaux d'effort de test unitaire des classes.

7.5.1 L'analyse de régression logistique binaire univariée

Comme définie dans la section 3.4.1.3, la régression logistique binaire prédit une variable indépendante en construisant une fonction logistique de probabilité à partir des variables indépendantes. Cette sous-section va se concentrer sur la possibilité de prédire 2 niveaux d'efforts de test. Pour ce faire, nous avons regroupé les 5 catégories issues des deux approches de classifications MEAN et KMEAN présentées plus haut, en deux clusters requérant: (1) un effort de test unitaire relativement faible auquel nous avons affecté la modalité 0, et (2) un effort de test unitaire relativement élevé auquel nous avons affecté la modalité 1. Ainsi, nous avons assigné aux catégories 4 et 5 (E et TE) dans la modalité 1 et les catégories restantes la modalité 0. Les résultats de cette classification pour chaque système sont donnés par la figure 17 suivante, selon les classifications MEAN et KMEAN. Les distributions des classes tournent autour de 30-70% dans les modalités 1 - 0 pour tous les systèmes dans l'approche de classification MEAN. Pour le KMEAN, les systèmes LUCENE et POI font exception avec 16.67% - 83.33% pour LUCENE et 11.40 - 88.60% pour POI.

	ANT	JFC	JODA	IO	IVY	LUCENE	MATH	POI
Mod-1	36,94%	34,96%	36,84%	32,84%	29,47%	29,82%	37,29%	34,19%
Mod-0	63,06%	65,04%	63,16%	67,16%	70,53%	70,18%	62,71%	65,81%

	ANT	JFC	JODA	IO	IVY	LUCENE	MATH	POI
Mod-1	28,83%	28,83%	30,26%	31,34%	26,32%	16,67%	32,20%	11,40%
Mod-0	71,17%	71,17%	69,74%	68,66%	73,68%	83,33%	67,80%	88,60%

Figure 17: Distribution des classes logicielles suivant les modalités binaires selon MEAN et KMEAN.

Nous avons émis les hypothèses suivantes pour toute métrique m_s du code source logiciel:

- H1: Une classe logicielle avec une grande valeur de métrique m_s (faible valeur de Qi) a tendance à requérir un effort de test plus élevé qu'une classe ayant une faible valeur de m_s (grande valeur de Qi).
- H0: Une classe logicielle avec une grande valeur de métrique m_s (faible valeur de Qi) n'a pas tendance à requérir un effort de test plus élevé qu'une classe ayant une faible valeur de m_s (grande valeur de Qi).

Nous avons appliqué la régression logistique univariée avec comme variable explicative les métriques logicielles OO et les Qi (pris séparément) et comme variable dépendante binaire les modalités obtenues après regroupement des 5 catégories. Nous avons déterminé les coefficients de Nagelkerk R^2, le *2log*, le coefficient normalisé *b*, ainsi que la *valeur-p* associée afin d'estimer le niveau de significativité du modèle de régression et de celui de l'apport de la variable indépendante (la métrique). Les performances sur les données d'entrainement de la régression ont été évaluées grâce à l'aire sous la courbe ROC (Section 3.4.1.3). Rappelons qu'une valeur plus grande que 0.70 veut dire que le modèle

est bon [113], alors qu'une valeur proche de 0.5 indique qu'il ne fait pas mieux que le modèle aléatoire. L'objectif ici n'est pas de construire un modèle de prédiction, mais de déterminer l'effet de chaque métrique sur l'effort de test unitaire. Les résultats de l'analyse sont présentés dans les tables 34 pour la classification MEAN et 35 pour KEMAN.

Table 34: Résultats de la régression logistique univariée pour MEAN.

		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	R^2	27.52%	9.88%	10.36%	24.50%	4.68%	2.53%	0.13%	27.53%
	2Log	< 0.0001	0.004	0.005	< 0.0001	0.061	0.1701	0.76	< 0.0001
	b	-0.593	0.318	0.002	0.587	0.007	0.231	-0.018	0.631
	p-value	< 0.0001	0.005	0.064	< 0.0001	0.071	0.1747	0.781	< 0.0001
	AUC	0.805	0.687	0.713	0.814	0.692	0.6014	0.539	0.812
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JFC	R^2	25.15%	8.71%	17.22%	23.51%	0.97%	0.39%	0.77%	18.48%
	2Log	< 0.0001	0	< 0.0001	< 0.0001	0.223	0.44	0.279	< 0.0001
	b	-0.563	0.296	0.001	0.748	0.001	0.096	0.028	0.543
	p-value	< 0.0001	0	0.001	< 0.0001	0.218	0.438	0.273	< 0.0001
	AUC	0.766	0.652	0.758	0.772	0.64	0.574	0.624	0.754
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JODA	R^2	38.60%	21.48%	56.56%	45.33%	21.72%	7.87%	10.64%	20.71%
	2Log	< 0.0001	0	< 0.0001	< 0.0001	0.001	0.045	0.019	0
	b	-0.815	0.506	0.002	1.017	0.013	0.395	-15.161	0.62
	p-value	< 0.0001	0.001	0	< 0.0001	0.001	0.047	0.994	0.005
	AUC	0.823	0.757	0.931	0.838	0.804	0.772	0.831	0.807
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IO	R^2	54.38%	42.92%	61.71%	61.83%	34.98%	20.85%	4.26%	58.28%
	2Log	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.002	0.165	< 0.0001
	b	-1.217	1.261	0.053	3.954	0.066	-1.038	0.253	3.001
	p-value	< 0.0001	0	0	< 0.0001	0.002	0.009	0.195	0
	AUC	0.885	0.841	0.896	0.915	0.815	0.792	0.566	0.907
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IVY	R^2	16.25%	1.22%	0.00%	4.69%	30.24%	19.89%	0.01%	7.41%
	2Log	0.001	0.367	0.96	0.075	< 0.0001	0	0.928	0.024
	b	-0.434	0.108	0	0.213	0.016	0.573	0.007	0.271
	p-value	0.001	0.363	0.96	0.083	0	0.001	0.927	0.031
	AUC	0.722	0.638	0.682	0.716	0.859	0.77	0.483	0.717
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
LUCENE	R^2	16.41%	7.18%	13.05%	11.56%	9.30%	0.00%	0.65%	14.98%
	2Log	0	0.015	0.002	0.002	0.009	0.971	0.497	0
	b	-1.389	0.266	0.001	0.545	0.01	0.009	-0.027	0.607
	p-value	0.008	0.02	0.107	0.017	0.017	0.971	0.545	0.011
	AUC	0.704	0.64	0.588	0.66	0.719	0.506	0.493	0.7
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
MATH	R^2	34.61%	25.31%	24.99%	34.61%	44.64%	32.02%	6.53%	34.22%
	2Log	< 0.0001	0.001	0.001	< 0.0001	< 0.0001	0	0.099	< 0.0001
	b	-0.755	0.623	0.009	1.323	0.094	-1.416	-0.931	1.175
	p-value	0.001	0.004	0.006	0.003	0.001	0.004	0.276	0.003
	AUC	0.791	0.784	0.674	0.792	0.859	0.802	0.641	0.811
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
POI	R^2	22.47%	15.92%	5.40%	21.65%	13.55%	5.25%	0.10%	14.67%
	2Log	< 0.0001	< 0.0001	0	< 0.0001	< 0.0001	0	0.639	< 0.0001
	b	-0.587	0.49	0.001	0.744	0.013	-0.384	-0.024	0.527
	p-value	< 0.0001	< 0.0001	0.004	< 0.0001	< 0.0001	0.001	0.677	< 0.0001
	AUC	0.738	0.711	0.54	0.734	0.709	0.633	0.563	0.725

Table 35: Résultats de la régression logistique univariée pour KMEAN.

		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	R^2	23.90%	13.30%	13.00%	26.40%	5.00%	0.30%	0.10%	28.50%
	2Log	< 0.0001	0.001	0.001	< 0.0001	0.046	0.63	0.81	< 0.0001
	b	-4.316	0.08	0.003	0.035	0.007	0.075	-0.013	0.007
	p-value	< 0.0001	0.002	0.022	< 0.0001	0.059	0.63	0.819	< 0.0001
	AUC	0.804	0.714	0.781	0.828	0.699	0.543	0.529	0.828
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JFC	R^2	24.40%	9.60%	11.20%	22.70%	1.20%	0.00%	0.70%	20.20%
	2Log	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.164	0.79	0.313	< 0.0001
	b	-4.09	0.038	0.001	0.021	0.001	0.033	0.026	0.004
	p-value	< 0.0001	0	0.006	< 0.0001	0.161	0.789	0.307	< 0.0001
	AUC	0.776	0.664	0.708	0.786	0.619	0.54	0.585	0.782
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JODA	R^2	60.30%	32.00%	49.40%	64.70%	14.80%	0.20%	4.80%	32.60%
	2Log	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.004	0.764	0.106	< 0.0001
	b	-9.548	0.17	0.002	0.071	0.01	0.055	-0.246	0.006
	p-value	< 0.0001	0	0.001	< 0.0001	0.005	0.763	0.283	0.001
	AUC	0.916	0.815	0.925	0.944	0.763	0.6	0.779	0.912
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IO	R^2	58.60%	39.70%	47.40%	66.40%	27.20%	22.00%	4.50%	61.40%
	2Log	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0	0.001	0.142	< 0.0001
	b	-10.004	0.334	0.043	0.218	0.055	-0.983	0.276	0.041
	p-value	0	0	0.001	0	0.004	0.005	0.198	0
	AUC	0.905	0.841	0.825	0.933	0.769	0.764	0.552	0.916
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IVY	R^2	10.90%	0.80%	0.10%	2.10%	18.70%	17.50%	0.10%	4.70%
	2Log	0.007	0.474	0.801	0.237	0	0	0.856	0.077
	b	-2.415	0.01	0	0.007	0.011	0.51	-0.013	0.002
	p-value	0.008	0.468	0.811	0.232	0.001	0.001	0.861	0.079
	AUC	0.685	0.638	0.326	0.664	0.772	0.742	0.445	0.672
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
LUCENE	R^2	15.30%	5.30%	13.50%	11.00%	5.80%	0.40%	0.00%	15.50%
	2Log	0.001	0.055	0.002	0.005	0.045	0.625	0.889	0.001
	b	-17.913	0.039	0.001	0.012	0.007	0.118	-0.005	0.003
	p-value	0.016	0.048	0.077	0.04	0.049	0.621	0.892	0.014
	AUC	0.69	0.609	0.613	0.649	0.682	0.556	0.455	0.677
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
MATH	R^2	45.10%	20.50%	18.40%	44.30%	37.40%	31.80%	8.60%	43.20%
	2Log	< 0.0001	0.002	0.004	< 0.0001	< 0.0001	< 0.0001	0.052	< 0.0001
	b	-7.618	0.247	0.007	0.1	0.079	-1.275	-1.102	0.023
	p-value	0	0.008	0.015	0.001	0.001	0.002	0.212	0.001
	AUC	0.846	0.754	0.597	0.849	0.816	0.768	0.572	0.862
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
POI	R^2	21.10%	9.20%	10.80%	23.30%	11.60%	6.80%	0.20%	18.10%
	2Log	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0	0.544	< 0.0001
	b	-6.564	0.039	0.001	0.027	0.01	-0.54	-0.058	0.004
	p-value	< 0.0001	< 0.0001	0	< 0.0001	< 0.0001	0.001	0.655	< 0.0001
	AUC	0.751	0.662	0.58	0.736	0.634	0.661	0.554	0.74

L'analyse basée sur la classification MEAN montre (table 34) que les modèles univariés issus des métriques Qi et LOC sont tous significatifs. Les performances des modèles varient globalement d'un système à un autre. En analysant le R^2, le 2log, le coefficient b (normalisé) et sa *p-value*, on constate que les modèles issus des Qi sont plus significatifs et performants que ceux construits sur les données de la métrique LOC. En effet, sur les 8 systèmes, les modèles de LOC ne performent significativement mieux (que

Qi) que sur le système IO. La métrique WMC produit des modèles significatifs pour 7 systèmes sur 8. IVY fait figure d'exception pour la métrique de complexité où le 2log et la *valeur-p* ne sont pas significatifs (>5%).

Dans la table 35, les résultats d'analyses logistiques dont la variable dépendante est basée sur le KMEAN montrent que les indicateurs de qualité sont la seule métrique à proposer des modèles significatifs pour tous les systèmes. LOC et WMC échouent à construire des modèles significatifs pour IVY alors que RFC et DIT le font. Le fait que ces deux métriques soient capturées par les indicateurs de qualité ([107,130]) peut expliquer les performances des Qi pour ce système. Nous constatons, ainsi, l'avantage qu'ont les Qi de capturer de manière intégrée le flux de contrôle, la complexité, le couplage dans une même métrique. Pris séparément, chaque métrique échoue (au moins dans le cas d'un système) à prédire la testabilité des classes, qui selon plusieurs chercheurs est une caractéristique externe, complexe et dépendante de plusieurs facteurs. Le Qi, en combinant plusieurs métriques dans sa formulation, tire avantage de la multiplicité des facteurs qu'il capture et permet de bâtir des modèles logistiques significatifs. Nous pouvons, au vu de ces résultats, rejeter H0 et garder H1 pour Qi, LOC et WMC pour tous les systèmes.

Pour comparer les modèles des métriques logicielles pour les classifications MEAN et KMEAN, nous avons représenté dans les figures 18 et 19 les différences (Δ) des coefficients R^2 et AUC des 8 systèmes pour les modèles de Qi, LOC et WMC (les plus significatifs).

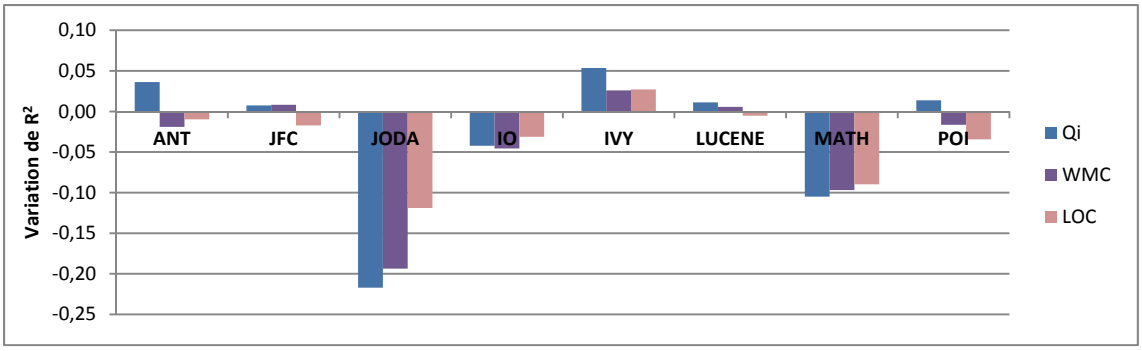


Figure 18: Variation de R^2 entre KMB et MB: $\Delta(R^2) = R^2(MB) - R^2(KMB)$.

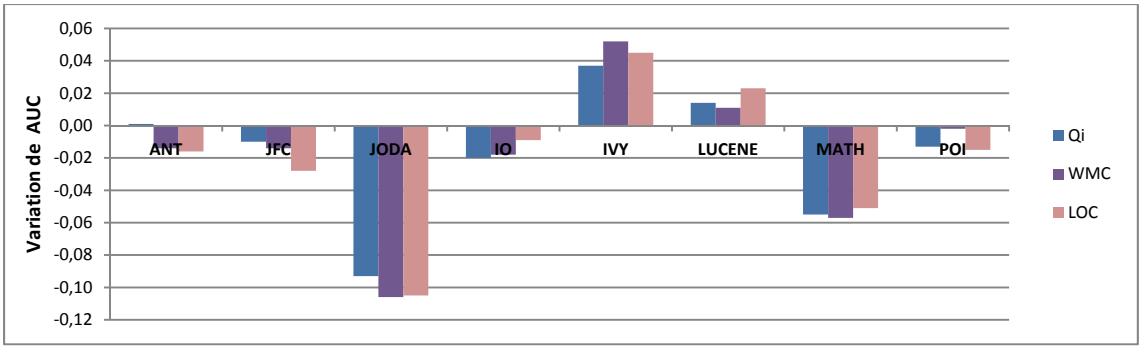


Figure 19: Variation de AUC entre KMB et MB: Δ(AUC) = AUC(MB) – AUC(KMB).

- Pour les 2 modèles logistiques univariés basés sur la métrique Qi, les performances sont comparables à 5% près dans 5 systèmes (les différences Δ(AUC) et Δ(R^2) plus petites que 5%). Les systèmes IVY, JODA et MATH font exception et les valeurs de R^2 varient respectivement de 10.90% à 10.25% (IVY), de 38.60% à 60.27% (JODA) et de 45.13% à 34.61% (MATH).

- Pour les 2 modèles logistiques univariés basés sur la métrique WMC, les performances sont comparables, sauf pour les systèmes JODA et MATH. Dans ces deux cas, les valeurs de R^2 varient respectivement de 64.71% à 45.33% (JODA) et de 44.26% à 34.61% (MATH).

- Pour les 2 modèles logistiques univariés basés sur la métrique LOC, les performances sont comparables, sauf pour les systèmes JODA et MATH. Dans ces deux cas, le R^2 varie respectivement de 32.55% à 20.71% (JODA) et de 43.18% à 34.22% (MATH).

Nous constatons que les performances des modèles des 3 métriques CBO, WMC et LOC pour la classification MEAN sont, en général, relativement plus faibles que celles de la classification KMEAN pour les mêmes métriques.

7.5.2 L'analyse de régression linéaire univariée

Nous utilisons, dans cette section, la régression linéaire (cf. section 3.4.1.2) pour évaluer l'effet des métriques logicielles sur les 5 niveaux d'effort de test unitaire. Nous voulons vérifier une des hypothèses suivantes:

- H0: Une classe logicielle avec une grande valeur de métrique m_s (faible valeur de Qi) n'a pas tendance à requérir un niveau d'effort de test plus élevé qu'une classe ayant une faible valeur de m_s (grande valeur de Qi).

- H1: Une classe logicielle avec une grande valeur de métrique m_s (faible valeur de Qi) a tendance à requérir un niveau d'effort de test plus élevé qu'une classe ayant une faible valeur de m_s (grande valeur de Qi).

La variable dépendante sera représentée par les 5 niveaux issus de la classification KMEAN. En effet, l'ordonnancement de ces clusters avait été étudié en fonction des moyennes des métriques. Nous avions résolu 3 conflits en priorisant la métrique TLOC, et les 3 solutions trouvées correspondaient à la tendance croissante de (presque) toutes les métriques de test (cf. figure 15). Nous avons naturellement affecté les valeurs 1 à 5 aux catégories 1 à 5. Forts de cette relation d'ordre dans les catégories, nous appliquons ici la régression linéaire. Nous l'évaluerons grâce au coefficient R^2, la pente *b* de la droite de régression (et sa normalisée bêta), ainsi que sa *p-value*. Nous avons aussi déterminé l'ordonnée à l'origine (intercept) pour caractériser la droite de régression.

La table 36 résume les résultats obtenus pour les 8 systèmes logiciels. Elle montre que les seuls modèles de régression linéaire univariée qui sont significatifs pour tous les systèmes (selon les *valeurs-p* des coefficients b) sont issus des métriques LOC et Qi et que les performances (des modèles issus) des métriques varient d'un système à un autre. La régression linéaire basée sur le Qi est relativement plus significative que les autres en termes de lien avec l'effort de test unitaire. Ce modèle est plus performant sur 6 systèmes (sur un total de 8). Le coefficient négatif de la pente indique que les niveaux d'effort ordonnés de 1 à 5 (du TF à TE) varient inversement avec le Qi, ce qui est normal vu la formulation de ce dernier. Ici, comme dans le cas de la régression logistique binaire, les différents niveaux du système IVY sont les moins bien prédits par les modèles de régression. Cependant, la métrique RFC y obtient les meilleurs résultats (R^2 = 14.63%, b significatif), suivie de Qi (R^2=10.47%, b significatif). Ces deux métriques ayant en commun la capture du flux de contrôle, nous pouvons déduire que dans le cas d'IVY l'effort est relativement lié au flux de contrôle, ce qui pourrait expliquer les faibles performances des

métriques restantes.

Table 36: Résultats de l'analyse de la régression linéaire univariée KMEAN.

		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	Intercept	3.835	1.43	0.096	1.427	0.046	-0.007	-0.008	1.389
	R^2	24.48%	16.72%	10.40%	25.51%	5.46%	0.21%	0.14%	27.59%
	beta	-0.502	0.418	0.322	0.5117	0.234	0.046	0.037	0.532
	b	-2.591	0.057	0.001	0.019	0.005	0.04	0.01	0.004
	p-value	< 0.0001	< 0.0001	0.001	< 0.0001	0.014	0.634	0.698	< 0.0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JFC	Intercept	3.219	1.654	0.155	1.557	0.001	-0.004	0.015	1.555
	R^2	24.39%	8.98%	15.91%	26.15%	0.55%	0.09%	1.89%	23.88%
	beta	-0.497	0.306	0.399	0.515	0.074	0.03	0.137	0.492
	b	-2.024	0.02	0	0.009	0	0.025	0.026	0.002
	p-value	< 0,0001	< 0,0001	< 0,0001	< 0,0001	0.267	0.655	0.039	< 0,0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
JODA	Intercept	3.82	1.03	0.487	0.888	0.242	0.04	-0.004	1.555
	R^2	46.12%	23.85%	49.43%	52.83%	25.22%	5.27%	0.91%	19.40%
	beta	-0.684	0.499	0.703	0.731	0.502	0.229	-0.095	0.44
	b	-3.6547	0.1	0.001	0.027	0.01	0.25	-0.04	0.002
	p-value	< 0,0001	< 0,0001	< 0,0001	< 0,0001	< 0,0001	0.046	0.413	< 0,0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IO	Intercept	4.308	1.532	0.231	1.617	0.415	0.11	0.029	1.55
	R^2	69.22%	44.36%	24.30%	49.64%	42.43%	12.35%	4.37%	53.06%
	beta	-0.835	0.672	0.493	0.71	0.651	-0.351	0.209	0.733
	b	-3.541	0.114	0.001	0.02	0.025	-0.327	0.153	0.005
	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.004	0.089	< 0.0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
IVY	Intercept	2.77	1.734	-0.008	1.742	0.137	0.06	-0.007	1.5981
	R^2	10.42%	0.49%	0.27%	1.21%	14.63%	6.99%	0.36%	7.18%
	beta	-0.337	0.124	-0.052	0.15	0.382	0.264	-0.06	1.598
	b	-1.453	0.009	0	0.004	0.006	0.187	-0.02	0.002
	p-value	0.001	0.23	0.618	0.146	0	0.01	0.562	0.005
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
LUCENE	Intercept	5.143	1.464	0.207	1.496	0.173	-0.009	-0.006	1.478
	R2	14.60%	11.49%	21.39%	23.13%	18.02%	0.02%	0.31%	28.60%
	beta	-0.392	0.35	0.463	0.488	0.424	0.015	-0.056	0.541
	b	-3.512	0.028	0	0.007	0.006	0.013	-0.007	0.001
	p-value	< 0.0001	0	< 0.0001	< 0.0001	< 0.0001	0.877	0.555	< 0.0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
MATH	Intercept	4.824	1.622	0.246	1.456	0.332	0.278	0.004	1.398
	R^2	51.38%	16.91%	25.86%	43.70%	34.31%	29.04%	2.15%	42.94%
	beta	-0.723	0.428	0.509	0.668	0.586	-0.539	-0.147	0.663
	b	-4.0667	0.145	0.005	0.029	0.038	-0.567	-0.075	0.007
	p-value	< 0,0001	0.001	< 0,0001	< 0,0001	< 0,0001	< 0,0001	0.268	< 0,0001
		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
POI	Intercept	3.808	1.318	0.074	1.228	0.114	0.071	-0.003	1.248
	R^2	20.13%	10.38%	7.71%	23.34%	11.62%	7.38%	0.03%	19.28%
	beta	-0.451	0.326	0.278	0.485	0.341	-0.272	-0.016	0.442
	b	-2.668	0.019	0	0.01	0.004	-0.173	-0.004	0.002
	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.766	< 0.0001

7.5.3 L'analyse de régression logistique univariée multinomiale

En utilisant les valeurs moyennes des métriques de test pour regrouper les classes testées en 5 clusters selon l'effort de construction et d'écriture qu'elles requièrent (MEAN), les catégories 1 et 5 sont clairement ordonnées, car les moyennes des métriques sont toutes supérieures dans la catégorie 5 et toutes inférieures dans la catégorie 1. Cela n'est pas le cas dans les catégories intermédiaires. En effet, en vérifiant 3 conditions sur 4, rien ne garantit que les moyennes des métriques de test de la catégorie 4 sont toutes supérieures à celles de la catégorie 3 ou 2 et vice-versa. L'ordre est plutôt qualitatif, car c'est le nombre de conditions imposées sur les métriques qui s'accroit de la catégorie 1 à la catégorie 5. Cela fait des données de cette classification les candidates idéales pour l'analyse logistique multinomiale (MLR) exposée dans la section 3.4.1.4. Il s'agit d'une généralisation de la régression logistique binaire dans le cas où la variable dépendante à plus de 2 états qualitatifs (réponses). La MLR permet de comparer chaque catégorie à une catégorie de base, la catégorie 1 (TF) pour nous. Les prédicteurs (variables indépendantes) vont être pris tour à tour sur l'ensemble des métriques logicielles incluant Qi. Nous avons déterminé le coefficient R^2 de Nagelkerk et 2log pour analyser la significativité par rapport au modèle trivial ainsi que les coefficients b des catégories (2, 3, 4, 5) et leurs niveaux de significativité (*p-value*) par rapport à la catégorie de base (catégorie 1). La table 37 montre les indicateurs obtenus après analyse. Nous interprétons, dans ce qui suit, les résultats pour chaque système:

- Pour ANT, la faible valeur de 2log (<5%) des modèles multinomiaux basés sur Qi, CBO, LCOM, WMC, DIT et LOC indique que ces 6 mesures améliorent significativement le modèle de base (basé sur le plus grand nombre de cas). Les métriques restantes (RFC, DIT et NOC) n'apportent pas d'amélioration significative par rapport au modèle de base pour être interprétées. Leurs modèles ne sont pas significatifs dans le cas d'ANT. Comme suggéré par leurs coefficients R^2, les variances expliquées par les modèles multinomiaux basés sur Qi, LOC et WMC sont supérieures à celles fournies par les autres métriques. Notons également que les coefficients R^2 de Qi et LOC sont les plus élevés (et sont comparables) et sont nettement supérieurs à celui de WMC.

Table 37: Résultats de l'analyse de régression multinomiale.

		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
ANT	R^2	0.267		0.138		0.105		0.238		0.046		0.118		0.06		0.258	
	2Log	< 0,0001		0.004		0.021		< 0,0001		0.295		0.011		0.164		< 0,0001	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-0.63	0.033	0.461	0.04	0.467	0.462	0.609	0.064	0.181	0.378	0.471	0.005	-1.09	0.09	0.609	0.063
		-1.109	0.001	0.64	0.011	0.884	0.168	1.068	0.002	0.266	0.26	0.094	0.655	-0.315	0.584	1.099	0.002
		-1.051	0.001	0.608	0.017	1.009	0.11	1.023	0.003	0.374	0.095	0.51	0.015	-0.883	0.334	1.087	0.002
		-1.255	0	0.866	0.001	1.131	0.074	1.245	0	0.405	0.079	0.44	0.051	-0.091	0.609	1.277	0
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
JFC	R^2	0.255		0.1		0.139		0.243		0.012		0.009		0.026		0.202	
	2Log	< 0,0001		0		< 0,0001		< 0,0001		0.645		0.765		0.241		< 0,0001	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-0.162	0.241	0.038	0.777	0.105	0.866	0.228	0.343	-0.122	0.342	-0.009	0.94	0.208	0.21	0.107	0.602
		-0.328	0.02	0.086	0.537	-0.247	0.768	0.435	0.058	-0.039	0.764	0.112	0.362	0.264	0.097	0.254	0.193
		-0.543	< 0,0001	0.27	0.024	1.012	0.008	0.803	< 0,0001	0.064	0.585	0.128	0.279	0.187	0.288	0.523	0.001
		-0.912	< 0,0001	0.467	< 0,0001	1.22	0.001	1.074	< 0,0001	0.089	0.402	0.039	0.724	0.259	0.095	0.798	< 0,0001
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
JODA	R^2	0.447		0.298		0.429		0.435		0.339		0.133		0.094		0.214	
	2Log	< 0,0001		< 0,0001		< 0,0001		< 0,0001		< 0,0001		0.047		0.153		0.003	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-0.421	0.288	0.778	0.044	-0.404	0.803	0.606	0.211	0.874	0.014	0.422	0.173	-28.99	0.997	0.743	0.034
		-0.421	0.065	0.389	0.062	0.541	0.286	0.835	0.006	0.717	0.002	0.359	0.063	-0.043	0.816	0.722	0.008
		-1.174	0.077	-0.038	0.94	1.362	0.012	1.505	0.001	0.355	0.457	-0.029	0.955	-28.99	0.998	0.903	0.007
		-1.65	< 0,0001	0.876	0	1.598	0	1.324	< 0,0001	0.905	< 0,0001	0.465	0.008	-28.99	0.994	0.793	0.003
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
IO	R^2	0.587		0.489		0.523		0.65		0.396		0.154		0.053		0.635	
	2Log	< 0,0001		< 0,0001		< 0,0001		< 0,0001		< 0,0001		0.034		0.497		< 0,0001	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-1.258	0.001	1.507	0.007	15.2	0.058	4.368	0.001	1.23	0.006	-0.033	0.859	-0.246	0.586	3.504	0.001
		-1.507	0.001	1.422	0.026	17.907	0.035	5.597	0	1.067	0.047	-0.163	0.53	0.092	0.731	4.049	0.001
		-1.941	0	2.249	0	26.766	0.001	6.972	< 0,0001	1.411	0.006	-0.674	0.068	0	1	5.459	< 0,0001
		-2.337	< 0,0001	2.561	< 0,0001	27.893	0.001	7.522	< 0,0001	1.969	< 0,0001	-0.674	0.025	0.244	0.179	5.981	< 0,0001
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
IVY	R^2	0.171		0.036		0.035		0.047		0.23		0.161		0.058		0.073	
	2 Log	0.003		0.53		0.541		0.373		0		0.004		0.269		0.157	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-0.294	0.096	0.193	0.227	-0.924	0.382	0.142	0.431	0.281	0.139	0.234	0.182	0.312	0.257	0.072	0.718
		-0.305	0.144	0.001	0.997	-1.464	0.415	0.176	0.382	0.151	0.522	0.177	0.404	-24.058	0.993	0.14	0.517
		-0.548	0.029	0.18	0.414	-0.046	0.866	0.222	0.309	0.69	0.007	0.716	0.006	0.266	0.373	0.26	0.227
		-0.648	0.001	0.235	0.138	-0.022	0.893	0.307	0.052	0.796	0	0.528	0.004	0.103	0.774	0.375	0.017
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
LUCENE	R^2	0.233		0.097		0.171		0.18		0.186		0.1		0.076		0.196	
	2Log	< 0,0001		0.029		0.001		0		0		0.026		0.079		0	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-2.487	0.004	0.196	0.236	2.502	0.261	0.739	0.027	0.714	0.006	0.109	0.464	-0.217	0.328	0.921	0.015
		-3.191	0	0.202	0.303	3.248	0.135	0.789	0.033	0.815	0.003	0.473	0.005	-0.467	0.296	1.209	0.001
		-2.293	0.014	0.28	0.105	2.804	0.209	0.726	0.048	0.762	0.005	0.246	0.131	-1.555	0.196	0.911	0.023
		-3.148	0	0.54	0.002	3.917	0.071	1.241	0.001	0.991	0	-0.239	0.406	0.037	0.805	1.309	0.001
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
MATH	R^2	0.508		0.302		0.275		0.525		0.422		0.294		0.075		0.489	
	2Log	< 0,0001		0.001		0.002		< 0,0001		< 0,0001		0.001		0.369		< 0,0001	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-1.106	0.008	0.793	0.02	0.519	0.121	2.289	0.007	0.848	0.024	-0.425	0.159	-0.542	0.614	1.753	0.012
		-0.285	0.508	1.056	0.002	-15.037	0.346	0.289	0.791	0.479	0.205	-0.425	0.159	-0.08	0.781	0.618	0.449
		-1.436	0.001	1.011	0.001	0.72	0.016	2.889	0	1.201	0.001	-1.152	0.013	-0.869	0.44	2.213	0.001
		-1.903	< 0,0001	0.662	0.051	0.739	0.016	3.29	0	1.568	0	-1.014	0.027	-19.928	0.993	2.737	0
		Qi		CBO		LCOM		WMC		RFC		DIT		NOC		LOC	
POI	R^2	0.263		0.189		0.155		0.268		0.195		0.071		0.006		0.197	
	2Log	< 0,0001		< 0,0001		< 0,0001		< 0,0001		< 0,0001		< 0,0001		0.748		< 0,0001	
		Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value	Value	P-value
	b	-0.495	0	0.568	0	1.987	0.001	0.759	0	0.612	0	-0.137	0.089	-0.014	0.851	0.599	0
		-0.656	< 0,0001	0.756	< 0,0001	2.146	0	0.982	< 0,0001	0.739	< 0,0001	-0.326	0.002	-0.31	0.465	0.757	< 0,0001
		-0.687	< 0,0001	0.638	< 0,0001	0.051	0.963	0.916	< 0,0001	0.574	0.003	-0.402	0	-0.217	0.533	0.591	0.002
		-1.108	< 0,0001	0.946	< 0,0001	2.369	< 0,0001	1.463	< 0,0001	1.088	< 0,0001	-0.288	0.007	-0.017	0.862	1.063	< 0,0001

Les coefficients b et leurs *valeurs-p* calculés pour les catégories 2, 3, 4 et 5 suggèrent que Qi et CBO sont les métriques les plus sensibles aux différents niveaux d'effort d'écriture et de construction des tests unitaires. En fait, les métriques Qi et CBO ont toutes leurs *valeurs-p* inférieures à 5% pour tous les niveaux d'effort de tests unitaires. Cependant, les métriques WMC et LOC ne parviennent pas à discerner correctement la catégorie 2 de la catégorie 1. Rappelons que le niveau 1 est celui sur lequel toutes les valeurs des métriques de test sont de petite taille (plus petite que la valeur moyenne) et les classes dans la catégorie 2 ont une seule des valeurs des 4 métriques de test supérieure à la valeur moyenne. Les deux premières catégories sont plus difficiles à distinguer l'une de l'autre. Enfin, on remarque que DIT distingue correctement le niveau 2 du niveau 1.

- Pour JFC, contrairement à ANT, la métrique LCOM améliore de manière significative le modèle de base (2Log <0.0001 <α = 5%), tandis que DIT échoue (2log = 0.765> α) à améliorer le modèle de base. Pour JFC comme pour ANT, les métriques RFC et NOC n'améliorent pas le modèle de base. En fait, toutes leurs valeurs de 2log sont supérieures à α. Les métriques Qi, CBO, WMC et LOC gardent la même tendance que pour ANT: elles ont toutes amélioré le modèle logistique multinomial de base et expliquent significativement la variance. Qi et WMC ont les valeurs les plus élevées pour le coefficient R^2 suivies LOC, LCOM et CBO. Pour les deux systèmes, la métrique Qi apparaît à la première place en termes de variance expliquée. Les coefficients b suggèrent que cette métrique (Qi) ne parvient pas à différencier le niveau 1 du niveau 2 (valeur p = 0.241 > α), mais discerne le niveau 1 des autres niveaux. Dans le cas de JFC, aucune autre métrique ne parvient à discerner le niveau 1 des niveaux 2 et 3. Les métriques CBO, LCOM, WMC et LOC discernent avec précision les niveaux 4 et 5 du niveau 1.
- Pour JODA, les valeurs 2log de toutes les métriques (à l'exception NOC) montrent une amélioration significative par rapport au modèle de base. On observe des valeurs plus élevées de coefficient R^2 de Nagelkerk par rapport aux systèmes précédents (ANT et JFC). Ici aussi, Qi et WMC ont les meilleures explications de la variance (0.447 et 0.435). LCOM suit avec R^2 = 0.429. Notons que la métrique LOC distingue toutes les catégories, tandis que WMC et RFC échouent pour les catégories 2 et 4 (respectivement). Même si la métrique Qi

offre la meilleure valeur de R^2, elle ne discerne que le niveau 5 du niveau 1. Ce résultat peut être expliqué par la distribution des classes logicielles dans les catégories 1-5. En effet, lors de l'analyse de la distribution du système JODA dans la table 32, nous avions constaté que la distribution ne compte en réalité que 3 catégories significatives. Les catégories 2 et 4 étant négligeables en termes d'observations. D'autre part, le nombre d'observations des catégories 2, 3 et 4 réunies est relativement faible (21% des classes du système JODA, la plus faible) comparativement aux autres systèmes dans lesquels ces catégories intermédiaires regroupent au moins 30% des observations. Dans la section 6.4 précédente, nous avons observé, après avoir effectué l'ACP pour le système JODA, que les métriques de tests unitaires sont fortement corrélées entre elles et ont tendance à se regrouper dans la première composante (plus de 80% de l'information). Dans ce cas, toutes les valeurs des métriques de test varient dans le même sens et envoient ainsi la plupart des classes tests dans les niveaux extrêmes 1 et 5 avec notre classification MEAN (les métriques sont inférieures ou supérieures aux moyennes en même temps, car étant fortement et positivement corrélées).

- Pour IO, comme pour JODA, seule la métrique NOC n'apporte pas d'amélioration significative au modèle de base (0.497> α = 5%). Ainsi, NOC sera ignorée dans les interprétations suivantes. Avec des coefficients R^2 respectivement de 0,65 et 0,635, les métriques WMC et LOC fournissent les meilleurs pouvoirs explicatifs des variances des catégories, suivies par Qi (0.587), LCOM (0.523), CBO (0.489), RFC (0.396) et DIT (0.154). Tous les niveaux sont correctement discernés du niveau 1 par les métriques Qi, CBO, WMC, RFC, et LOC. Notons que LCOM ne discerne pas le niveau 2 du niveau 1, tandis que DIT ne discerne que le niveau 5 du niveau 1.
- Pour IVY, les valeurs de R^2 sont inférieures à celles des systèmes précédents. Par ailleurs, seules les métriques Qi, RFC et DIT fournissent une amélioration significative. RFC offre le meilleur taux d'explication de la variance des catégories (R^2 = 0.230), suivie par Qi et DIT. Les trois métriques distinguent aussi les catégories 4 et 5 de la catégorie 1. Toutes les autres métriques échouent à discerner les cinq catégories. Rappelons que la métrique RFC compte le nombre total des invocations de méthodes directes et indirectes en réponse

à un appel de méthode, et DIT la profondeur de l'arbre d'héritage d'une classe. Les deux dimensions fournies par ces métriques, ainsi que la complexité (WMC) et le couplage sont déjà intégrés dans la formulation des indicateurs de qualité. Cela peut expliquer pourquoi la métrique Qi fournit toujours une amélioration significative du modèle de base lorsque l'une de ces métriques est significative.

- Pour LUCENE, tous les indicateurs fournissent des améliorations significatives au modèle de base, sauf la métrique NOC. Dans le cas de LUCENE, la métrique Qi présente la valeur la plus élevée de R^2 (0.233). La métrique Qi est suivie respectivement par LOC (0.196), RFC (0.186), WMC (0.180), LCOM (0.171), DIT (0.100) et CBO (0.097). Selon les valeurs-p des coefficients b, les modèles générés par les métriques Qi, WMC, RFC et LOC, distinguent nettement les 5 niveaux d'effort de test unitaire. CBO et DIT discernent respectivement le $5^{ème}$ et le $3^{ème}$ niveau.
- Pour MATH, les métriques WMC, Qi et LOC ont les plus grandes valeurs de R^2. Seule NOC échoue à fournir un modèle significativement diffèrent du modèle de base. Qi, WMC, RFC et LOC ne discernent pas correctement la catégorie 3 de la catégorie 1. En plus de ne pas discerner la catégorie 3, LCOM et DIT échouent également de distinguer la catégorie 2 de la catégorie 1. Seule CBO semble correctement distinguer la catégorie 3, mais elle échoue pour la catégorie 5.
- Pour POI, les métriques WMC et Qi affichent les plus grandes valeurs de R^2. Elles sont suivies par LOC (0.197), RFC (0.195), CBO (0.189), LCOM (0.155) et DIT (0.071). Dans ce cas aussi, la métrique NOC est la seule qui ne présente pas d'amélioration significative ($2\log = 0.748 > \alpha = 5\%$). Le reste des métriques discernent correctement les différentes catégories, mis à part LCOM qui échoue pour la catégorie 4 et DIT pour la catégorie 2.

En résumé, les résultats obtenus tendent à montrer que la métrique Qi est plus sensible aux différentes catégories (niveaux d'effort de test) des systèmes. Ce résultat peut être expliqué par le fait que cette métrique intègre les différentes dimensions capturées par les autres attributs. La métrique NOC ne fournit aucune amélioration significative du modèle de base quel que soit le système considéré

7.6 Résumé des résultats

Les tables 38 et 39 donnent les résumés respectifs de l'analyse de régression logistique binaire, de l'analyse de régression linéaire et de l'analyse de régression logistique multinomiale. Nous indiquons dans ces tables, pour chaque système, l'ordre (rang) en termes de performance du modèle univarié issu de chaque métrique. Nous avons utilisé des valeurs allant de 1 (pour indiquer que le modèle est le plus précis) à 8 (pour indiquer que le modèle est le moins précis). Les cases de couleur dans les tables indiquent que le modèle correspondant n'est pas significatif. Les dernières lignes de chaque table indiquent les valeurs moyennes des rangs de chaque modèle selon les types de classifications. La dernière ligne donne une idée sur la performance globale de chaque métrique (moyenne de tous les rangs).

La table 38 nous fait constater que, dans l'ensemble, les modèles logistiques univariés basés sur la métrique Qi (pour les deux approches de catégorisation MEAN et KMEAN utilisées pour classer les classes testées) sont les plus précis en termes de prédiction de la variable dépendante (niveau d'effort de test unitaire) par rapport aux autres modèles univariés. La valeur moyenne des rangs du modèle est la plus petite, et ce pour les deux classifications. Le modèle basé sur la métrique Qi est suivi par celui basé sur la métrique WMC puis par celui basé sur la métrique LOC. En outre, comme nous pouvons le voir dans la table 38: (1) les modèles de régression logistique univariée basés respectivement sur les métriques Qi et LOC sont les seuls modèles significatifs pour tous les systèmes analysés, (2) les modèles de régression logistique univariée basés respectivement sur les métriques CBO, LCOM et WMC ne sont pas significatifs dans le cas du système IVY, (3) le modèle de régression logistique univariée basé sur RFC est non significatif pour la classification issue de MEAN pour les systèmes ANT et JFC, (4) les modèles de régression logistique univariée basés sur les métriques d'héritage DIT et NOC sont soit non significatifs, soit les moins performants, quand ils le sont. Enfin, si l'on considère uniquement les 2 modèles univariés qui sont significatifs pour tous les systèmes (celui basé sur Qi et celui basé sur LOC), et que l'on compte le nombre de fois où le modèle de régression logistique univariée est soit le plus précis ou le deuxième plus précis en termes de distinction des catégories, le modèle de

régression logistique univariée basé sur le Qi totalise 10 cas, et celui basé sur la métrique LOC totalise 5 cas. Donc, nous pouvons raisonnablement conclure que, pour l'analyse de régression logistique univariée binaire, et selon les deux approches que nous avons suivies dans la catégorisation des classes testées, le modèle basé sur la métrique Qi est le plus précis, suivi par les modèles basés sur les métriques LOC et WMC.

Table 38: Résumé des résultats de la régression logistique univariée binaire.

		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	MEAN	2	5	4	3	6	7	8	1
	KMEAN	3	4	5	2	6	7	8	1
JFC	MEAN	1	5	4	2	6	8	7	3
	KMEAN	1	5	4	2	6	8	7	3
JODA	MEAN	3	5	1	2	4	8	7	6
	KMEAN	2	5	3	1	6	8	7	4
IO	MEAN	4	5	2	1	6	7	8	3
	KMEAN	3	5	4	1	6	7	8	2
IVY	MEAN	3	6	8	5	1	2	7	4
	KMEAN	3	6	7	5	1	2	8	4
LUCENE	MEAN	1	6	3	4	5	8	7	2
	KMEAN	2	6	3	4	5	7	8	1
MATH	MEAN	2	6	7	2	1	5	8	4
	KMEAN	1	6	7	2	4	5	8	3
POI	MEAN	1	3	6	2	5	7	8	4
	KMEAN	2	6	5	1	4	7	8	3
MOYENNE	MEAN	2.13	5.13	4.38	2.63	4.25	6.5	7.5	3.38
	KMEAN	2.13	5.38	4.75	2.25	4.75	6.38	7.75	2.63
	Globale	2.13	5.25	4.56	2.44	4.5	6.44	7.63	3

En observant la table 39 suivante, on peut voir que, dans l'ensemble, le modèle logistique multinomial basé sur la métrique Qi (suivant la classification basée sur MEAN ou KMEAN) est le plus précis en termes de prédiction de la variable dépendante (niveau d'effort de test unitaire) par rapport aux autres modèles. Le modèle de régression linéaire simple basé sur la métrique WMC (selon les KMEAN) est le plus précis en termes de prédiction du niveau d'effort de test (par rapport aux autres métriques). Selon la valeur moyenne globale des rangs des modèles de régression, le rang moyen des modèles basés sur la métrique Qi est le plus petit. Il est suivi par le rang moyen des modèles basés sur WMC puis par ceux basés sur la métrique LOC.

Table 39: Résumé des résultats des régressions univariées linéaire et logistique multinomiales.

		Qi	CBO	LCOM	WMC	RFC	DIT	NOC	LOC
ANT	MEAN	1	4	6	3	8	5	7	2
	KMEAN	3	4	5	2	6	7	8	1
JFC	MEAN	1	5	4	2	7	8	6	3
	KMEAN	2	5	4	1	7	8	6	3
JODA	MEAN	1	5	3	2	4	7	8	6
	KMEAN	3	5	2	1	4	7	8	6
IO	MEAN	3	5	4	1	6	7	8	2
	KMEAN	1	4	6	3	5	7	8	2
IVY	MEAN	2	7	8	6	1	3	5	4
	KMEAN	2	6	8	5	1	4	7	3
LUCENE	MEAN	1	7	5	4	3	6	8	2
	KMEAN	5	6	3	2	4	8	7	1
MATH	MEAN	2	5	7	1	4	6	8	3
	KMEAN	1	7	6	2	4	5	8	3
POI	MEAN	2	5	6	1	4	7	8	3
	KMEAN	2	5	6	1	4	7	8	3
MOYENNE	MEAN	1.63	5.38	5.38	2.50	4.63	6.13	7.25	3.13
	KMEAN	2.38	5.25	5	2.13	4.38	6.63	7.5	2.75
	Globale	2.00	5.31	5.19	2.31	4.50	6.38	7.38	2.94

Par ailleurs, comme nous pouvons le voir dans la table 39, les modèles de régression linéaire univariée et de régression logistique multinomiale issus de la métrique Qi sont les seuls modèles significatifs pour tous les systèmes. Les modèles de régression logistique univariée basés respectivement sur les attributs CBO, LCOM, WMC et LOC ne sont pas significatifs dans le cas du système logiciel IVY. Les modèles de régression linéaire univariée et logistique multinomiale univariée basés sur les métriques d'héritage (DIT et NOC) ne sont généralement pas significatifs. Dans les rares cas où ils le sont, ils sont les moins performants.

Bien que les modèles de la métrique Qi soient dans l'ensemble plus précis (ou parmi les plus précis), il est tout de même important de remarquer que leurs performances varient d'un système à l'autre. Cela est également vrai pour les autres modèles issus des autres métriques. Nous avons voulu comprendre les raisons pour lesquelles un modèle est plus précis pour certains systèmes que pour d'autres, mais aussi (malgré le fait de leur significativité) les variations observées du R^2 correspondant (d'un système à l'autre). L'analyse approfondie du code source des classes testées de chaque système, et le code des classes de tests unitaires JUnit correspondantes, nous a permis de faire plusieurs observations qui peuvent en partie expliquer les raisons de ces différences:

- Les différences de taille entre les systèmes analysés: Les systèmes analysés varient en termes de taille, en particulier en ce qui concerne le nombre de classes dans le code source. POI est le plus grand des 8 systèmes analysés en termes de nombre de classes, et IO est le plus petit (avec 104 classes). En outre, des systèmes avec un nombre relativement petit de classes peuvent être importants en termes de nombre de lignes de code. Par exemple, le système JFC à moins de classes logicielles que ANT, LUCENE et IVY, mais en termes de lignes de code, JFC est plus grand. Ceci suggère que les classes de JFC contiennent en moyenne plus de lignes de code que celles d'ANT, LUCENE ou IVY.
- Les tests partiels des logiciels analysés: En observant la figure 13, on peut voir que dans tous les systèmes, les classes de test JUnit n'ont pas été développées pour toutes les classes logicielles. En outre, le nombre de classes pour lesquelles les tests unitaires JUnit ont été développés diffère également d'un système à l'autre. Le système qui a le plus grand nombre de classes couvertes par des tests JUnit est POI (404 classes).
- Le taux de classes logicielles explicitement testées: Le pourcentage de classes logicielles pour lesquelles les cas de tests JUnit ont été développés varie d'un système à l'autre. Il va de 15% pour ANT et IVY, à plus de 45% pour JFC (45,60%), MATH (59,20%), et IO (65,38%). Pour le plus grand des 8 systèmes (POI), ce taux n'est que de 26,25%.
- Le pourcentage de lignes de code source des classes logicielles explicitement testées : En analysant le code des cas de test JUnit de chaque système, nous avons constaté que le pourcentage de lignes de code source testées (nombre total de lignes de code pour lesquelles des cas de test JUnit ont été développés / nombre total de lignes de code source des classes) diffère également d'un système à l'autre. Il est compris entre 27% (pour ANT) et IO (82,85%). Pour POI, ce pourcentage est de 43,20%.
- Le nombre de lignes de test écrites pour une ligne de code logiciel: Nous avons aussi observé que le rapport entre le nombre de lignes de code de test et le nombre de lignes de code logiciel testées varie grandement suivant les systèmes. Dans certains cas, ce ratio est supérieur à 1 (JODA, MATH et IO), ce qui indique qu'il y a plus de lignes de test que de lignes de code testées. Dans le cas de LUCENE, ce ratio est proche de 1. Pour les systèmes restants, ce ratio est inférieur à 1.

- Enfin, les domaines d'application des logiciels: Les systèmes sont de domaines d'application très différents. En effet, tandis que certains comme JFC font intervenir des artéfacts objets complexes (designs complexes, héritage, polymorphismes, bibliothèque Swing de JAVA) d'autres comme MATH contiennent uniquement des algorithmes mathématiques fonctionnels et d'autres encore comme ANT utilisent beaucoup la réflexivité du langage (non capturée par l'analyse statique) pour son fonctionnement et des quantités de données « mock » (objets non-JAVA) lors des tests. Les classes des tests unitaires issues de ces différents systèmes ne peuvent structurellement pas être semblables, et l'effort de leur écriture et leur construction se répartissent dans différents facteurs qu'une métrique capturant un seul attribut peut difficilement appréhender.

Enfin, dans les deux types (tables 38 et 39) de regroupement, la moyenne globale des rangs indique relativement le même résultat: les modèles basés sur la métrique Qi sont les plus performants et sont suivis par WMC, LOC, RFC, LCOM, CBO, DIT et NOC.

7.7 Risque pour la validité

Certaines limitations liées aux conditions expérimentales de cette analyse peuvent restreindre les interprétations. Nous les détaillons dans cette section et proposons des pistes possibles pour réduire ces biais.

- Les risques pour la validité interne: Certains facteurs cachés peuvent influencer les résultats obtenus. En effet, dépendamment des stratégies de test adoptées, certaines classes ont vu les tests couvrir uniquement les méthodes complexes alors que les métriques logicielles (Qi et autres) couvrent toutes les méthodes des classes pour leurs calculs. Cela peut biaiser les relations entre les métriques des classes logicielles et les métriques des classes tests associées. En conséquence, les performances de certaines métriques peuvent artificiellement augmenter ou diminuer. Il serait judicieux de considérer le niveau de couverture de test des méthodes des classes logicielles en utilisant, dans les expérimentations, uniquement des classes logicielles avec de larges couvertures de tests unitaires des méthodes. Par ailleurs, la métrique LCOM ne peut être calculée pour toutes les classes de par sa définition (cf. section 3.2.1), ces classes ont été exclues des analyses ce

qui peut influer sur les performances. Il serait alors intéressant d'utiliser d'autres métriques de cohésion calculables pour toutes les classes logicielles comme celles de Henderson-Sellers [120]. Les variances observées dans les tables 28 et 29 peuvent impacter les modèles générés. Les classes logicielles peuvent être regroupées en clusters de faibles variances internes avant de construire les modèles. Ce qui permettrait là aussi de réduire le biais.

- Les risques pour la validité externe: Même si notre échantillon est assez large pour avoir des résultats significatifs, certains facteurs nous empêchent de généraliser les résultats observés. Le premier facteur est lié aux choix des classes logicielles explicitement testées. En effet, les statistiques montrent que pour tous les systèmes, les classes explicitement testées sont en général plus grandes et plus complexes. Qu'adviendra-t-il des résultats pour un système ne contenant que des classes de petite taille ? Un autre facteur pouvant limiter la généralisation des résultats est lié aux différents styles d'écriture adoptés par les développeurs lors de l'écriture des classes tests. Ces résultats comme nous l'avons vu précédemment peuvent impacter les métriques de test et par ricochet les résultats des classifications et des modèles de régression. Ces deux biais peuvent être réduits en considérant des systèmes pour lesquels un maximum de classes logicielles est explicitement testé et sélectionner les métriques de test les moins volatiles (comme TINVOK et TLOC) pour construire les niveaux d'effort. L'autre risque de validité externe concerne l'appariement des classes logicielles aux classes de tests unitaires JUnit. Ce risque a été discuté dans la section 5.5.

- Risques pour la validité conceptuelle: Les techniques de mesure utilisées dans notre démarche empirique peuvent ne pas saisir entièrement les concepts que l'on souhaite effectivement mesurer. En effet, l'effort d'écriture des classes tests concerne aussi la création de données qui ne sont pas des artefacts orientés objet. La métrique TDATA considérée dans l'expérimentation ne tient compte que des objets JAVA créés pour les tests et omet les données non-JAVA. Par ailleurs, la procédure d'appariement ignore certaines classes tests auxquelles nous n'avons pu associer aucune classe logicielle. Ces classes tests ont nécessité un effort d'écriture et de construction qui n'a pas été pris en

compte. D'autres classes de test unitaire sont réutilisées par héritage lors du test de plusieurs classes logicielles différentes. L'effort d'écriture et de construction de la classe de test est dans ce cas dédié à plusieurs classes, ce qui rend impossible sa circonscription à une seule classe logicielle. Effectuer une expérimentation dans un environnement contrôlé pourrait certes diminuer ce biais, mais éloignerait aussi les résultats de la réalité des contraintes du développement logiciel. Un autre risque pour validité conceptuelle est lié au processus de classification en catégories. Il s'agit, en fait, d'une discrétisation de la variable « effort » qui s'accompagne inévitablement de pertes d'informations. Le fait de fixer les catégories (ou niveau d'effort) à 5 peut entrainer des variances plus importantes dans les clusters finaux ou un débalancement de la répartition des classes logicielles dans les clusters. Il pourrait être judicieux là aussi de choisir des méthodes de classification basées sur le critère de la variance, ou la classification hiérarchique ascendante. Ces techniques détermineront automatiquement le nombre de clusters finaux en fonction des données.

7.8 Conclusion

L'étude que nous avons présentée analyse empiriquement le lien entre les métriques OO incluant les indicateurs de qualité et la testabilité des classes vue sous l'angle de différents niveaux d'efforts de test unitaire qu'elles requièrent. L'étude empirique couvre 8 systèmes logiciels *open source* de grande taille, écrits en JAVA pour lesquels des classes tests unitaires JUnit ont été développées pour certaines classes logicielles. L'effort requis pour l'écriture et la construction des tests unitaires JUnit a été mesuré grâce à 4 métriques de test puis regroupé selon les techniques MEAN et KMEAN en 5 niveaux. Nous avons utilisé les analyses de corrélation, de régression univariée (linéaire, logistique binaire et logistique multinomiale) pour étudier le lien entre Qi, métriques OO de CK incluant LOC et ces différents niveaux d'effort de test. L'objectif était d'étudier la capacité des métriques à prédire les différents niveaux d'effort. Les résultats ont montré que: (1) les métriques Qi, LOC et WMC prédisaient correctement les différents niveaux de test avec de meilleures performances pour les modèles issus des indicateurs de qualité, (2) les liens entre les modèles issus des autres métriques et les niveaux d'effort ont des significativités variables en fonctions des systèmes, et (3) la métrique NOC n'avait aucun pouvoir explicatif. Ces

travaux ont fait l'objet de publications [143,144].

Dans les 3 chapitres précédents, nous avons montré empiriquement l'existence de liens significatifs entre les indicateurs de qualité ainsi que les métriques logicielles avec l'effort de test unitaire du point de vue de l'écriture et de la construction des classes de tests unitaires. Dans le prochain chapitre, nous allons construire des modèles d'aide à l'orientation de l'effort global de test vers les composants critiques des systèmes logiciels à partir des métriques logicielles et des indicateurs de qualité.

CHAPITRE 8. INDICTEURS DE QUALITÉ ET ORIENTATION DE L'EFFORT DE TEST

8.1 Introduction

Après avoir démontré l'existence de liens significatifs entre les indicateurs de qualité et l'effort d'écriture et de création des tests unitaires, nous explorons dans ce chapitre, différentes approches permettant l'utilisation effective des indicateurs de qualité comme outil d'aide à l'orientation des tests. Plusieurs techniques d'optimisation des tests ont été proposées dans la littérature. Ces approches se focalisent surtout sur la priorisation des cas de test durant les tests de régression ou les tests fonctionnels. Elles supposent que les composants logiciels à tester sont déjà choisis, ce qui n'est pas trivial dans certaines situations. En effet, lors d'un processus de développement itératif, des groupes de fonctionnalités sont implémentés à chaque itération. Ces fonctionnalités regroupent plusieurs composants logiciels qui devront être testés unitairement. Sachant que le processus de test est gourmand en ressources, les responsables de projets choisissent alors de concentrer leurs ressources sur certains composants spécifiques. Il arrive, aussi, qu'au cours de la maintenance des changements introduits impactent un grand nombre de composants et rendent irréaliste tout test exhaustif. Dans cette situation également, les responsables de projet sont appelés à faire des choix sur des composants à tester.

Dans ce chapitre, nous abordons l'orientation de l'effort dans le cadre des tests unitaires des logiciels OO. Nous explorons la possibilité d'aider aux choix des composants logiciels à tester. L'objectif est de construire des modèles automatiques pouvant orienter l'effort global (dédié aux tests) vers les composants logiciels adéquats en nous basant uniquement sur les métriques de code source. Nous avons mené 3 séries d'expérimentations pour explorer les mécanismes d'orientation des tests à partir des indicateurs de qualité et de certaines métriques logicielles OO. Le premier protocole vise à caractériser les groupes de classes explicitement testées et les groupes de classes non explicitement testées des différents systèmes à partir de leurs attributs logiciels (incluant les Qi). Le deuxième protocole étudie les liens individuels des différents attributs logiciels

ainsi que les Qi, avec les choix effectués par les responsables de test. Et finalement, le troisième protocole étudie la possibilité d'apprendre automatiquement des choix des responsables des tests d'un système pour proposer des ensembles tests pour d'autres systèmes.

Afin d'alléger et de faciliter la lecture, nous utiliserons les expressions « responsable de test » pour designer tout développeur, équipe de développeurs, testeurs, ou gestionnaire de projet qui décident dans le cas d'un développement logiciel des classes qui seront explicitement testées. Rappelons qu'une classe logicielle explicitement testée est une classe pour laquelle il a été développé une classe test unitaire JUnit dédiée, que nous avons identifiée et appariée (section 3.3.1). Rappelons, par ailleurs, que pour un système, l'ensemble de test explicite est l'ensemble de classes explicitement testées. Il s'agit de l'ensemble formé des classes logicielles choisies par les responsables de test pour être explicitement testées.

8.2 Objectif

Réalisant que, dans la plupart des projets logiciels d'envergure, il est irréaliste d'effectuer une couverture de test complète, les responsables des tests sont souvent contraints de trouver un équilibre entre les ressources à pourvoir et les classes logicielles à tester pour maintenir la qualité de leur produit. L'exercice est difficile et consiste à cibler un nombre limité de classes logicielles qui seront testées explicitement. Pour être efficace, ce choix doit maximiser l'identification des fautes les plus critiques afin de pouvoir mettre (ou remettre) en production une version assez stable du logiciel. Ce choix constitue donc un enjeu économique et technique considérable et est souvent effectué de manière informelle selon les connaissances qu'ont les responsables des tests de leur système ainsi que leur expérience. Ce contexte met en lumière l'importance des outils d'aide et d'orientation des tests, et justifie l'objectif de recherche de ce chapitre. Nous explorons les liens entre les métriques logicielles et les ensembles tests explicites proposés par les responsables de test avant de proposer des modèles d'apprentissage pour orienter l'effort de test. Ainsi, nos investigations porteront sur le pouvoir explicatif et prédictif des Qi et

plus généralement les métriques OO sur le choix des classes explicitement testées. Le but final étant de construire un modèle automatique à partir de ces métriques qui permette d'orienter le choix des classes qui seront explicitement testées. Nous avons mis en place une démarche expérimentale visant à démontrer, d'une part, le lien entre les métriques et les classes logicielles explicitement testées dans un but d'analyse et de compréhension et, d'autre part, la possibilité d'utiliser ces métriques dans le but d'orienter l'effort global de test.

L'existence de liens significatifs suggèrera: (1) la possibilité de construire un outil apprenant capable d'aider à orienter l'effort global de test dans les logiciels, et (2) l'existence d'un certain nombre de critères (ou règles) informels qui conduiraient d'une manière directe ou indirecte les responsables à choisir les classes (candidates) qu'ils testeront explicitement. Ces règles seront, par ailleurs, liées aux attributs du code source des logiciels qui sont capturés par les métriques.

8.3 Métriques et systèmes considérés

Pour cette expérimentation, nous avons considéré 10 systèmes logiciels parmi ceux décrits dans la section 3.1. Il s'agit d'ANT, JFC, DBU, JODA, IO, POI, IVY, MATH, LOG4 et LUCENE. Pour ces 10 systèmes, certaines classes ont été unitairement testées. En appariant les classes logicielles aux classes tests, selon la méthode décrite à la section 3.3, nous avons associé la modalité 1 aux classes logicielles explicitement testées (ensembles tests proposés par les responsables de test) et la modalité 0 aux autres classes logicielles. Les critères de choix de ces classes nous sont (à priori) inconnus. Les logiciels que nous avons choisis sont de différentes tailles. Ils sont issus de domaines d'application tout aussi différents. Nous avons choisi les attributs LOC, CBO et WMC en plus des indicateurs de qualité (Qi) pour mener nos investigations. Ces métriques sont ressorties dans les sections précédentes comme étant liées à l'effort d'écriture et de construction des tests unitaires.

La table 40 résume les statistiques de ces métriques pour les 10 systèmes. Notons que pour ces investigations, des artéfacts triviaux comme les interfaces et les classes abstraites pures ont été retirées des données. La table 40 montre que les systèmes logiciels

considérés sont de différentes tailles. Les lignes de code vont de 7604 lignes réparties sur 100 classes logicielles pour le système IO à plus de 130 185 lignes dans 1382 classes logicielles pour le système POI.

Le nombre de classes et leur complexité cyclomatique varient tout autant. Les statistiques descriptives montrent 4 groupes de systèmes: (1) des systèmes de petite taille, autour de 100 classes comme IO et MATH, (2) des systèmes de taille moyenne autour de 200 classes comme LOG4J, DBU et JODA, (3) des systèmes (relativement) de grande taille entre 400 et 600 classes comme LUCENE, IVY, ANT et JFC, et (4) un système de très grande taille autour de 1000 classes avec POI.

Notons que la complexité cyclomatique moyenne varie beaucoup entre des systèmes de taille comparable. Par exemple, deux systèmes comme JODA et DBU avec un nombre similaire de classes (aux alentours de 200) ont des complexités cyclomatiques moyennes très différentes: 9,34 pour DBU et 31,18 pour JODA. LUCENE et JFC présentent également cette différence marquée des moyennes de complexité par classe, pour des systèmes de taille comparable. Dans notre jeu de données, chaque observation (classe logicielle) a, en plus des métriques Qi, CBO, LOC, et WMC, un attribut TESTED binaire prenant les modalités 1 ou 0 pour signifier qu'elle a été explicitement testée (1) ou pas (0).

Table 40: Statistiques descriptives des métriques des systèmes.

ANT	Qi	CBO	LOC	WMC
Obs.	663	663	663	663
Min.	0	0	1	0
Max.	1	41	1252	245
Som.	424.53	4613	63 548	12 034
μ	0.640	6.958	95.849	18.151
σ	0.355	7.250	132.915	24.168
Cv	0.555	1.042	1.387	1.332
DBU	Qi	CBO	LOC	WMC
Obs.	213	213	213	213
Min.	0	0	4	1
Max.	1	24	488	61
Som.	166.87	1316	12 187	1989
μ	0.783	6.178	57.216	9.338
σ	0.195	5.319	60.546	9.451
Cv	0.249	0.861	1.058	1.012
IO	Qi	CBO	LOC	WMC
Obs.	100	100	100	100
Min.	0	0	7	1
Max.	0.98	39	968	250
Som.	64.57	405	7604	1817
μ	0.646	4.050	76.040	18.170
σ	0.294	5.702	121.565	31.751
Cv	0.455	1.408	1.599	1.747
IVY	Qi	CBO	LOC	WMC
Obs.	610	610	610	610
Min.	0	0	2	0
Max.	1	92	1039	231
Som.	335.09	5205	50 080	9664
μ	0.623	8.533	82.098	15.843
σ	0.342	11.743	141.801	27.380
Cv	0.549	1.376	1.727	1.728
LOG4J	Qi	CBO	LOC	WMC
Obs.	231	231	231	231
Min.	0	0	5	1
Max.	0.98	107	1103	207
Som.	131.22	1698	20 150	3694
μ	0.568	7.351	87.229	15.991
σ	0.325	10.119	130.419	25.700
Cv	0.573	1.377	1.495	1.607

JFC	Qi	CBO	LOC	WMC
Obs.	411	411	411	411
Min.	0	0	4	0
Max.	1	101	2041	470
Som.	216.94	4861	67 481	13 428
μ	0.528	11.827	164.187	32.672
σ	0.363	14.066	228.056	46.730
Cv	0.688	1.189	1.389	1.430
JODA	Qi	CBO	LOC	WMC
Obs.	201	201	201	201
Min.	0	0	5	1
Max.	1	36	1760	176
Som.	68.24	1596	31 339	6269
μ	0.340	7.940	155.915	31.189
σ	0.351	6.443	210.974	30.553
Cv	1.035	0.811	1.353	0.980
POI	Qi	CBO	LOC	WMC
Obs.	1382	1382	1382	1382
Min.	0	0	2	0
Max.	1	168	1686	374
Som.	1051.31	9660	130 185	23 810
μ	0.761	6.990	94.200	17.229
σ	0.295	10.782	154.282	28.319
Cv	0.388	1.543	1.638	1.644
MATH	Qi	CBO	LOC	WMC
Obs.	94	94	94	94
Min.	0	0	2	0
Max.	1	18	660	174
Som.	36.65	306	7779	1824
μ	0.390	3.255	82.755	19.404
σ	0.302	3.716	97.601	25.121
Cv	0.774	1.141	1.179	1.295
LUCEN	Qi	CBO	LOC	WMC
Obs.	615	615	615	615
Min.	0	0	1	0
Max.	1	55	2644	557
Som.	568.63	3793	56 108	10 803
μ	0.925	6.167	91.233	17.566
σ	0.192	7.243	192.874	35.704
Cv	0.208	1.174	2.114	2.033

8.4 Classes logicielles explicitement testées et métriques logicielles

Nous investiguons les liens individuels entre les classes explicitement testées ou non explicitement testées avec leurs attributs (que sont les métriques logicielles incluant les Qi). Nous avons mené deux expérimentations distinctes pour analyser les liens. Dans un premier temps, nous avons analysé grâce au test Z, les moyennes des métriques logicielles pour les deux groupes. Nous avions, en effet, remarqué en analysant les statistiques descriptives des systèmes dans les expérimentations précédentes que les métriques des classes logicielles explicitement testées étaient en moyennes plus élevées que celles des

classes logicielles non explicitement testées. Cette première expérimentation permettra de confirmer de manière formelle ce constat. Dans un second temps, nous avons effectué une analyse de régression logistique univariée binaire. L'objectif est d'étudier le niveau de significativité de ces liens avec chaque métrique. Et finalement, nous avons utilisé les méthodes d'apprentissage automatique (régression logistique et réseau bayésien naïf), pour suggérer des ensembles de classes à tester explicitement pour un système, à partir de données de ses métriques de code source (incluant les Qi) et des connaissances acquises sur des systèmes différents. Le but, dans ce cas, étant de comprendre dans quelle mesure les critères de choix des classes à tester (s'ils existent) sont réutilisables dans le cadre de l'orientation des tests dans d'autres systèmes.

8.4.1 Le test Z de la moyenne

Nous avons utilisé le test de la moyenne décrit en 3.4.1.6 pour analyser les attributs logiciels des systèmes. Il s'agit d'un test permettant de déterminer, dans notre cas, si pour chaque système, le groupe de classes logicielles explicitement testées (modalité 1) a des moyennes (des métriques Qi, CBO, LOC et WMC) significativement différentes du groupe de classes logicielles non explicitement testées (modalité 0). L'objectif était de voir si ces métriques pouvaient significativement caractériser les 2 groupes de classes dans chaque système. Nous avons formulé les hypothèses nulle et alternative suivantes pour chaque système et chaque métrique logicielle m_s donnée:

- H0: La moyenne de la métrique m_s du groupe de classes logicielles de modalité 0 n'est pas significativement différente de la moyenne de la métrique m_s du groupe de classes de modalité 1.
- H1: La moyenne de la métrique m_s du groupe de classes logicielles de modalité 0 est significativement différente de la moyenne de la métrique m_s du groupe de classes de modalité 1.

Le test détermine une *valeur-p* que l'on compare au seuil de significativité typique $\alpha = 5\ \%$. Elle est la probabilité que la différence δ entre les moyennes μ des deux groupes soit nulle par hasard. Le coefficient Z fourni est comparé à un seuil critique de référence (Z/Cri) pour

une distribution normale. Nous avons aussi déterminé l'écart-type σ de chaque groupe. La table 41 présente les résultats obtenus pour les 10 systèmes analysés.

- Nous constatons que pour presque tous les systèmes (sauf IO), les moyennes des indicateurs de qualité des classes logicielles explicitement testées sont en moyenne significativement plus faibles que celles des classes non explicitement testées. Rappelons qu'une valeur faible de Qi (proche de 0) indique (entre autres) une classe complexe, fortement couplée avec les autres classes du système. Les résultats de test de la moyenne indiquent que les classes explicitement testées ont en moyenne un Qi plus faible que celles non explicitement testées. À l'exception du système IO pour lequel la différence des moyennes de Qi des deux groupes n'est pas significative: la *valeur-p* est de 0.054 (plus grand que α = 0.05). Nous pouvons, dans le cas de Qi, rejeter l'hypothèse nulle et garder l'hypothèse alternative pour tous les systèmes sauf IO.
- Nous observons les mêmes résultats pour la complexité et la taille. Les classes logicielles explicitement testées sont en moyenne significativement plus complexes (WMC) et de plus grande taille (LOC) que les classes logicielles non explicitement testées. Dans les résultats, le système MATH fait figure d'exception avec une *valeur-p* de 0.143 pour LOC et de 0.092 pour WMC, toutes les deux supérieures au seuil de significativité (α = 0.05). Nous pouvons, dans le cas de LOC et WMC, rejeter l'hypothèse nulle et garder l'hypothèse alternative pour tous les systèmes sauf MATH.
- La métrique CBO semble être la moins discriminante pour les modalités. En effet, les différences de moyennes entre les groupes explicitement testés et non explicitement testés ne sont pas significatives pour les systèmes LOG4J, MATH et IO. Ce qui peut s'expliquer entre autres par la plus grande visibilité de certains des attributs de code source logiciel. En effet la multiplicité de ligne de code (LOC) et la forte complexité cyclomatique (WMC) sont directement visibles dans le code source alors que le niveau de couplage entre les classes (capturé par CBO), particulièrement les liens entrants, n'est pas directement visible dans le code source d'une classe logicielle isolée. En fait, l'identification du couplage d'une classe (surtout entrant) nécessite une analyse plus approfondie du code source du système dans sa globalité.

Table 41: Test z de la moyenne.

ANT	Obs.	μ	σ	δ	Z	Z/Cri	p-value	JFC	Obs.	μ	σ	δ	Z	Z/Cri	p-value
Qi | 1	112	0.53	0.35	0.13	3.51	1.96	0.000	Qi | 1	229	0.38	0.34	0.13	3.51	1.96	0.000
Qi | 0	551	0.66	0.35					Qi | 0	182	0.71	0.31				
CBO | 1	112	10.41	8.60	-4.15	4.82	1.96	< 0,0001	CBO | 1	229	15.98	15.29	-0.40	-2.88	1.96	0.004
CBO | 0	551	6.26	6.75					CBO | 0	182	6.6	10.27				
LOC | 1	112	157.45	154.72	-74.12	4.77	1.96	< 0,0001	LOC | 1	229	232	273.2	-74.12	-4.77	1.96	< 0,0001
LOC | 0	551	83.33	124.64					LOC | 0	182	78.86	104.61				
WMC | 1	112	31.11	31.18	-15.59	5.05	1.96	< 0,0001	WMC | 1	229	46.28	57.17	-15.59	-5.05	1.96	< 0,0001
WMC | 0	551	15.52	21.61					WMC | 0	182	15.55	17.7				

DBU	Obs.	μ	σ	δ	Z	Z/Cri	p-value	JODA	Obs.	μ	σ	δ	Z	Z/Cri	p-value
Qi | 1	86	0.74	0.17	0.07	2.81	1.96	0.005	Qi | 1	76	0.23	0.33	0.18	3.71	1.96	0.000
Qi | 0	127	0.81	0.21					Qi | 0	125	0.41	0.35				
CBO | 1	86	8.78	5.48	-4.36	-6.14	1.96	< 0,0001	CBO | 1	76	10.62	7.34	-4.31	-4.47	1.96	< 0,0001
CBO | 0	127	4.42	4.46					CBO | 0	125	6.31	5.26				
LOC | 1	86	72.93	49.08	-26.36	-3.35	1.96	0.001	LOC | 1	76	231.89	279.65	-122.17	-3.55	1.96	0.000
LOC | 0	127	46.57	65.49					LOC | 0	125	109.72	138.41				
WMC | 1	86	11.41	7.17	-3.47	-2.86	1.96	0.004	WMC | 1	76	44.75	39.98	-21.81	-4.46	1.96	< 0,0001
WMC | 0	127	7.94	10.56					WMC | 0	125	22.94	19.11				

IO	Obs.	μ	σ	δ	Z	Z/Cri	p-value	POI	Obs.	μ	σ	δ	Z	Z/Cri	p-value
Qi | 1	66	0.61	0.30	0.11	1.93	1.96	0.054	Qi | 1	387	0.63	0.31	0.18	10.07	1.96	< 0,0001
Qi | 0	34	0.72	0.27					Qi | 0	995	0.81	0.27				
CBO | 1	66	4.68	6.27	-1.86	-1.73	1.96	0.083	CBO | 1	387	10.76	13.72	-5.24	-6.96	1.96	< 0,0001
CBO | 0	34	2.82	4.34					CBO | 0	995	5.52	8.99				
LOC | 1	66	95.85	144.34	-58.26	-3.08	1.96	0.002	LOC | 1	387	150.89	200.23	-78.74	-7.20	1.96	< 0,0001
LOC | 0	34	37.59	37.88					LOC | 0	995	72.15	125.65				
WMC | 1	66	22.45	37.82	-12.60	-2.50	1.96	0.013	WMC | 1	387	29.08	37.94	-16.46	-8.03	1.96	< 0,0001
WMC | 0	34	9.85	11.37					WMC | 0	995	12.62	21.91				

IVY	Obs.	μ	σ	δ	Z	Z/Cri	p-value	MATHS	Obs.	μ	σ	δ	Z	Z/Cri	p-value
Qi | 1	95	0.35	0.35	0.33	8.57	1.96	< 0,0001	Qi | 1	58	0.34	0.29	0.14	2.22	1.96	0.027
Qi | 0	515	0.68	0.31					Qi | 0	36	0.48	0.31				
CBO | 1	95	18.23	16.1	-11.49	-6.73	1.96	< 0,0001	CBO | 1	58	3.24	3.82	-0.04	-0.05	1.96	0.963
CBO | 0	515	6.74	9.78					CBO | 0	36	3.28	3.65				
LOC | 1	95	189.97	209.36	-127.77	-5.79	1.96	< 0,0001	LOC | 1	58	92.95	114.66	-26.61	-1.46	1.96	0.143
LOC | 0	515	62.2	115.33					LOC | 0	36	66.33	61.02				
WMC | 1	95	34.47	38.66	-22.07	-5.39	1.96	< 0,0001	WMC | 1	58	22.31	30.35	-7.59	-1.69	1.96	0.092
WMC | 0	515	12.41	23.24					WMC | 0	36	14.72	12.56				

LOG4J	Obs.	μ	σ	δ	Z	Z/Cri	p-value	LUCENE	Obs.	μ	σ	δ	Z	Z/Cri	p-value
Qi | 1	44	0.35	0.3	0.28	5.39	1.96	< 0,0001	Qi | 1	114	0.84	0.26	0.10	3.93	1.96	< 0,0001
Qi | 0	187	0.62	0.31					Qi | 0	501	0.94	0.17				
CBO | 1	44	8.41	8.1	-1.31	-0.90	1.96	0.366	CBO | 1	114	9.9	10.72	-4.59	-4.42	1.96	< 0,0001
CBO | 0	187	7.1	10.57					CBO | 0	501	5.32	5.89				
LOC | 1	44	175.27	176.85	-108.76	-3.91	1.96	< 0,0001	LOC | 1	114	193.8	340.6	-125.96	-3.89	1.96	0.000
LOC | 0	187	66.51	107.84					LOC | 0	501	67.88	128.7				
WMC | 1	44	34.16	43.04	-22.44	-3.40	1.96	0.001	WMC | 1	114	35.89	61.18	-22.49	-3.85	1.96	0.000
WMC | 0	187	11.72	17.19					WMC | 0	501	13.4	25.06				

Ces résultats préliminaires montrent que les moyennes des métriques logicielles des deux groupes de classes sont significativement distinctes. Les métriques logicielles considérées (particulièrement Qi, LOC et WMC) caractérisent les groupes de classes testées explicitement et les groupes de classes non explicitement testées. Cela suggère l'existence de liens entre les attributs (métriques logicielles) et les critères de choix des classes

logicielles que les responsables de test se proposent de tester explicitement. Nous explorons ces liens individuellement dans la section suivante.

8.4.2 L'analyse de régression logistique binaire univariée

Après avoir caractérisé les groupes de classes explicitement testées et non explicitement testées par les attributs logiciels, nous présentons dans cette section l'étude empirique que nous avons menée pour évaluer les liens individuels de chaque métrique de code source (LOC, WMC et CBO ainsi que les Qi) avec le statut (testée explicitement ou non) des classes logicielles. Pour ce faire, nous avons utilisé la régression logistique univariée décrite à la section 3.4.1.3 afin d'analyser les liens entre les métriques OO considérées, ainsi que les Qi, et la variable TESTED formée des modalités binaires classes testées (1) et classes non testées (0). Nous avons analysé chaque modèle logistique obtenu à travers les différents coefficients caractéristiques que nous avons calculés. Nous avons utilisé les valeurs normalisées du coefficient b pour comparer l'importance relative des différentes métriques dans l'explication de la variable binaire TESTED. Nous avons calculé les coefficients R^2 de Nagelkerk pour déterminer le pouvoir explicatif du modèle de régression obtenu par rapport au modèle de base (basé sur la distribution naturelle de la variable binaire dépendante). Nous avons aussi calculé l'aire sous la courbe ROC (AUC) pour déterminer le niveau d'ajustement du modèle avec les données. Un modèle est dit bon si son AUC est supérieure à 0.70 [113]. Les résultats sont résumés dans la table 42.

L'analyse de la table 42 nous permet d'observer 3 tendances globales selon les systèmes:

- Les résultats obtenus avec ANT et JODA ont les coefficients R^2 de Nagelkerk significatifs pour toutes les métriques (*valeurs-p* < 5%), avec une valeur d'AUC < 70 %. La valeur significative de R^2 traduit le fait que l'apport informationnel des métriques améliore significativement le modèle de base (basé sur la moyenne de la variable binaire). Les coefficients b sont aussi tous significatifs au regard de leurs *valeurs-p* respectives. Cependant, les modèles issus de ces métriques ne constituent pas de bons prédicteurs (AUC < 0.70). Ce résultat peut s'expliquer par le faible taux de couverture (au niveau, classe) des tests explicites dans le cas d'ANT (16.9%), tandis que pour JODA, l'explication

pourrait se trouver dans le faible taux de tests explicites parmi les classes complexes (la complexité cyclomatique moyenne des classes de JODA non explicitement testées est de 22.94 contre 15.52 pour ANT). Enfin, pour les deux systèmes, un des facteurs clés du choix des classes qui ont été explicitement testées pourrait être la complexité cyclomatique (WMC) des classes. Les modèles logistiques qui en sont issus ont les R^2 (7.7 % pour ANT et 16 % pour JODA) les plus élevés.

- Les résultats obtenus avec les systèmes IO et MATH présentent des valeurs de R^2 et b non significatives ainsi qu'une capacité prédictive faible (AUC < 70 %) pour les modèles issus de la plupart des métriques. Pour IO, le facteur taille (LOC), fait figure d'exception et présente de bonnes performances (R^2 significatif de 15,4 %, b significatif de 1.053 et AUC = 0.710 > 0.7). La taille semble être un des facteurs qui explique le choix des classes logicielles explicitement testées. Le signe positif des coefficients b montre que les classes de plus grande taille ont été testées. L'indice AUC > 0.70 montre que le modèle issu de LOC s'ajuste aux données, autrement dit, la plupart des classes de plus grande taille ont été testées. Le rapport moyen de taille des classes explicitement testées vs celles non explicitement testées est d'un facteur de 2.55 avec la cote Z du test de la moyenne la plus élevée en absolu (3.08). Ce qui confirme les performances du facteur taille. Pour MATH, seule la métrique Qi présente des coefficients significatifs (R^2 et b), avec cependant un AUC < 70, les modèles issus des autres métriques ne performent pas. Ce résultat peut s'expliquer, entre autres, par le fait que les choix des classes à tester a fait intervenir, dans le cas de MATH, plusieurs facteurs et qu'au bout du compte, une partie des classes retenues durant le processus de sélection a été explicitement testée. Ce résultat est confirmé par la cote Z de Qi qui bien qu'étant significative reste faible (<3).

Table 42: Régressions logistiques univariées.

		Qi		CBO		LOC		WMC	
		valeurs	p-value	valeurs	p-value	valeurs	p-value	valeurs	p-value
ANT	R^2	0.029	0.001	0.066	< 0,0001	0.057	< 0,0001	0.077	< 0,0001
	b	-0.189	0.001	0.264	< 0,0001	0.238	< 0,0001	0.284	< 0,0001
	AUC	0.632		0.667		0.684		0.694	
JFC	R^2	0.298	< 0,0001	0.168	< 0,0001	0.235	< 0,0001	0.253	< 0,0001
	b	-0.619	< 0,0001	0.532	< 0,0001	0.999	< 0,0001	1.124	< 0,0001
	AUC	**0.779**		**0.723**		**0.768**		**0.771**	
DBU	R^2	0.045	0.008	0.214	< 0,0001	0.062	0.002	0.043	0.009
	b	-0.207	0.009	0.533	< 0,0001	0.266	0.004	0.207	0.012
	AUC	0.671		**0.762**		**0.756**		**0.755**	
JODA	R^2	0.088	0.000	0.138	< 0,0001	0.116	< 0,0001	0.160	< 0,0001
	b	-0.311	0.000	0.392	< 0,0001	0.430	0.001	0.456	< 0,0001
	AUC	0.685		0.670		0.690		0.679	
IO	R^2	0.048	0.059	0.041	0.083	0.154	0.001	0.095	0.008
	b	-0.235	0.072	0.269	0.145	1.053	0.010	0.762	0.049
	AUC	0.623		0.620		**0.710**		0.684	
POI	R^2	0.099	< 0,0001	0.065	< 0,0001	0.071	< 0,0001	0.097	< 0,0001
	b	-0.314	< 0,0001	0.284	< 0,0001	0.307	< 0,0001	0.389	< 0,0001
	AUC	**0.728**		0.686		**0.739**		**0.756**	
IVY	R^2	0.202	< 0,0001	0.160	< 0,0001	0.131	< 0,0001	0.105	< 0,0001
	b	-0.516	< 0,0001	0.417	< 0,0001	0.369	< 0,0001	0.332	< 0,0001
	AUC	**0.752**		**0.792**		**0.805**		**0.779**	
MATH	R^2	0.069	0.026	0.000	0.963	0.027	0.167	0.035	0.115
	b	-0.264	0.030	-0.005	0.963	0.193	0.221	0.241	0.187
	AUC	0.634		0.533		0.557		0.546	
LOG4J	R^2	0.165	< 0,0001	0.004	0.465	0.136	< 0,0001	0.149	< 0,0001
	b	-0.466	< 0,0001	0.061	0.450	0.419	0.000	0.471	0.000
	AUC	**0.734**		0.599		**0.786**		**0.799**	
LUCENE	R^2	0.051	< 0,0001	0.079	< 0,0001	0.086	< 0,0001	0.085	< 0,0001
	b	-0.213	< 0,0001	0.284	< 0,0001	0.366	< 0,0001	0.391	< 0,0001
	AUC	**0.729**		0.657		**0.749**		**0.750**	

- Pour JFC, DBU, POI, IVY, LOG4J et LUCENE, nous observons des coefficients R^2, b et les scores AUC significatifs. Dans ces systèmes, le choix des tests est significativement expliqué par les métriques logicielles considérées. Nous noterons aussi, au regard des résultats précédents, que les cotes Z de ces systèmes sont particulièrement élevées. Ce résultat peut s'expliquer par la multiplicité des critères de choix pour l'orientation des tests ou la forte corrélation entre les métriques logicielles pour ces systèmes. Dans le groupe, JFC et IVY se distinguent par des performances significatives de toutes les métriques. Un maximum de 29.8 % pour R^2 est obtenu pour le système JFC avec la métrique Qi. Dans les systèmes JFC et IVY, toutes les métriques que nous avons considérées semblent avoir été prises en compte lors de l'orientation des tests (du moins les attributs qu'elles capturent). Notons que le couplage est la métrique la moins performante de ce groupe de résultats.

Cet attribut semble le moins pris en considération dans l'orientation des tests. Notons finalement que dans 5 des 6 systèmes de ce groupe (à l'exception de LUCENE), les Qi offrent les meilleures performances en termes de R^2. La métrique synthétique Qi qui incorpore différentes métriques logicielles dont la taille, le couplage et la complexité, semble mieux performer quand les critères de choix pour l'orientation des tests sont multiples.

Globalement, l'analyse de la régression logistique suggère que les critères qui orientent les choix des gestionnaires quant aux classes logicielles à tester explicitement peuvent, dans la plupart des cas, être expliqués par le couplage, la complexité cyclomatique, la taille, et surtout par des métriques synthétiques comme les Qi.

8.5 Apprentissage automatique et validation inter-systèmes

Dans cette section, nous avons utilisé des algorithmes d'apprentissage sur des données des systèmes pour proposer des ensembles tests sur d'autres systèmes logiciels. L'objectif de cette expérimentation est de voir dans quelles mesures les critères utilisés par les responsables pour décider des classes logicielles qui seront explicitement testées, peuvent être réutilisés sur des systèmes différents. En d'autres termes, cela revient à étudier la transposabilité des critères de choix des responsables des tests d'un système à un autre. L'hypothèse est la suivante:

Hypothèse: En considérant les valeurs des métriques Qi, CBO, LOC et WMC pour l'ensemble des classes logicielles d'un système S_i, pour lequel les responsables ont proposé des ensembles tests, il est possible de construire automatiquement un apprenant capable de proposer, pour un autre système S_j, un ensemble de test « proche » de l'ensemble test qu'auraient proposé les responsables des tests de S_j.

La validation de cette hypothèse suggèrerait l'existence d'attributs des classes logicielles qui déterminent les choix des responsables des tests. Il sera alors possible, à partir de ces critères (attributs) de construire des outils automatiques d'aide à l'orientation des tests.

Dans cette analyse, nous avons choisi deux méthodes d'apprentissage. La régression logistique et le classificateur naïf bayésien (décrits à la section 3.4). La régression logistique est particulièrement adaptée à la prédiction de variables ayant une modalité binaire et permet, en plus du taux d'erreur, d'analyser éventuellement plusieurs paramètres de sortie pouvant expliquer les performances de l'apprenant. Le classificateur naïf bayésien assume certes une forte hypothèse sur les données, mais est particulièrement efficace et rapide pour les grandes masses de données. Il requiert relativement peu de données d'entrainement, ce qui est un avantage lorsque l'entrainement se fait sur des systèmes de petite taille. Les données de chacun des 10 systèmes serviront tour à tour de données d'entrainement et de données de test pour les deux méthodes d'apprentissage. Rappelons que pour ces systèmes logiciels, nous avons les ensembles tests proposés par les responsables des tests issus d'équipes différentes et les systèmes ont aussi des domaines d'application différents. La matrice de la table 43 donne, dans chaque case, le taux de bonnes classifications des méthodes d'apprentissage s'entrainant sur les données du système à la ligne i et dont la validation est effectuée sur les données du système à la colonne j. La diagonale (k, k) donne l'ajustement des méthodes d'apprentissage sur les données du système k. Nous interprétons les performances des méthodes d'apprentissage en analysant les résultats selon les données d'entrainement (lignes) et les données de test (colonnes).

Pour les lignes, l'analyse des données d'entrainement permet de dégager 3 groupes de systèmes. Rappelons que chaque ligne représente le système dont les données ont servi d'ensemble d'entrainement pour la méthode d'apprentissage indiquée.

- Les systèmes issus du groupe {IO, IVY, LOG4J, MATH} produisent de mauvais jeux d'entrainement. En effet, les apprenants issus de ces jeux de données n'ont aucun pouvoir prédictif sur les ensembles explicites de tests des autres systèmes. Ce qui peut s'expliquer soit par un faible nombre d'observations dans les données d'entrainement, soit par leur mauvaise distribution, soit par un surapprentissage ou soit encore par des critères de choix non capturés par les métriques Qi, CBO, LOC et WMC. Dans le cas de MATH, nous voyons clairement en regardant la diagonale (9,9), qu'il ne s'agit pas d'un problème de

surapprentissage, mais que les métriques ne capturent pas les critères des gestionnaires. En conséquence, aucun autre apprenant ne prédit non plus de manière satisfaisante les données du système MATH (voir la colonne de MATH-9). La non capture des critères de choix par les métriques ainsi que la faible taille des observations de MATH (94 observations, la plus faible du groupe) font qu'il n'est pas possible d'obtenir un bon apprenant sur ses données (voir la ligne de 9, aucun taux de classement n'égale ou ne dépasse 0.7).

Table 43: Validations croisées des apprentissages.

		JFC	ANT	DBU	IO	IVY	JODA	LOG4J	LUCEN	MATH	POI
JFC	LOG	0,81	0,589	0,611	0,525	0,682	**0,704**	**0,812**	0,699	0,625	0,687
	BNAIF	**0,819**	0,677	**0,74**	0,626	**0,779**	0,695	**0,778**	**0,711**	0,5	**0,748**
ANT	LOG	0,666	0,684	0,646	0,691	**0,745**	0,669	0,479	0,676	0,506	0,683
	BNAIF	**0,77**	0,717	**0,704**	0,578	**0,757**	0,668	**0,762**	**0,715**	0,548	**0,736**
DBU	LOG	**0,732**	0,682	0,778	0,525	**0,767**	**0,722**	0,687	**0,701**	0,486	**0,717**
	BNAIF	**0,762**	0,637	0,843	0,665	0,688	0,556	**0,75**	0,696	0,544	**0,755**
IO	LOG	0,668	0,625	0,623	0,831	**0,77**	0,501	0,586	0,609	0,602	0,625
	BNAIF	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5
IVY	LOG	0,689	0,639	0,632	0,644	0,803	0,684	0,651	0,615	0,553	0,663
	BNAIF	**0,726**	0,649	0,695	0,694	0,822	0,652	**0,715**	**0,709**	0,526	**0,72**
JODA	LOG	0,697	**0,7**	**0,721**	0,477	**0,764**	0,745	**0,7**	**0,715**	0,463	**0,705**
	BNAIF	**0,712**	0,607	0,575	0,525	**0,733**	0,771	0,688	**0,711**	0,577	0,655
LOG4J	LOG	0,692	0,579	0,496	0,547	0,484	0,677	0,87	0,671	0,591	0,594
	BNAIF	**0,79**	0,67	0,665	0,594	**0,742**	0,657	0,838	**0,752**	0,558	**0,753**
LUCENE	LOG	**0,776**	0,658	**0,735**	0,509	**0,745**	**0,743**	**0,798**	0,738	0,57	**0,731**
	BNAIF	**0,765**	0,653	**0,709**	0,596	**0,743**	**0,711**	**0,766**	0,758	0,566	**0,736**
MATH	LOG	0,385	0,487	0,31	**0,716**	0,46	0,3	0,41	0,307	0,651	0,444
	BNAIF	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5
POI	LOG	**0,752**	0,666	0,675	0,601	**0,784**	**0,744**	**0,754**	**0,717**	0,574	0,727
	BNAIF	**0,779**	0,673	**0,762**	0,609	**0,74**	0,692	**0,781**	**0,758**	0,503	0,791

LOG4J et IO semblent présenter un problème de surapprentissage en observant la diagonale en (4,4) et (7,7). En effet, les apprenants qui s'entrainent sur les deux systèmes présentent d'excellents taux de classement sur leurs propres jeux de données, mais ne peuvent prédire aucun jeu de données des autres systèmes (voir ligne 4 et 7). La taille significative de LOG4J (231 observations) réconforte cette hypothèse tandis que pour IO la petite taille (100 observations) peut expliquer, en partie, les mauvais résultats de la ligne 4. En termes de test logiciel (pour LOG4J et éventuellement pour IO), cela signifie que les responsables des tests ont pris en compte des caractéristiques très spécifiques au système en plus de celles capturées par les métriques pour choisir les classes à tester explicitement. Le cas d'IVY est particulier dans ce groupe. Il peut simplement s'expliquer par un mauvais balancement des observations. En effet, seuls 15.6 % des 610 classes logicielles ont été explicitement testées. La sensibilité du classificateur naïf de Bayes au surapprentissage (qui

est observé pour les systèmes IO et MATH) peut expliquer les faibles taux de bonnes classifications (0.5) observées sur les données de test issues des autres systèmes dans les lignes 4 et 9 de la table 43.

- Pour les systèmes du groupe {JFC, ANT}, les apprenants qui s'entrainent sur leurs données classifient de manière satisfaisante au moins 5 autres systèmes. Les performances des apprenants s'entrainant sur JFC sont légèrement supérieures à celles des apprenants d'ANT. Ce constat peut s'expliquer, entre autres, part une meilleure répartition des observations côté JFC. En effet, les responsables des tests de JFC ont développé des classes de test pour environ une classe logicielle sur deux, contre 17 % pour ANT. Ils ont choisi des classes selon des critères qui sont capturés (en partie) par les métriques et transposables à d'autres systèmes.
- Pour les systèmes du groupe {DBU, JODA, LUCENE, POI} les apprenants qui s'y entrainent ont une très bonne capacité de prédiction sur les autres systèmes (hormis MATH). Les lignes 3, 6, 8 et 10 de la table 44 présentent des taux de classement en général supérieurs à 70 %. Dans ce groupe, LUCENE et POI constituent des ensembles d'entrainement particulièrement bons pour les algorithmes d'apprentissage. En termes de test logiciel, les responsables des tests de ces systèmes ont choisi les classes qui seront explicitement testées selon des critères qui sont capturés (en grande partie) par les métriques et qui sont transposables à la plupart des autres systèmes.

Selon les colonnes (données de test), l'analyse des performances suggère 3 groupes de données d'entrainement formés par les systèmes. Rappelons que chaque colonne représente un système dont les données ont servi d'ensemble test (validation) pour l'algorithme indiqué.

- Les systèmes du groupe {JFC, IVY, LOG4J, POI} (colonnes vert foncé) sont en général bien prédits par les apprenants qui se sont entrainés sur les différents systèmes (mis à part MATH). En plus de s'expliquer par les caractéristiques des données d'entrainement, ces performances sont aussi liées aux caractéristiques des données de test, autrement dit sur les critères de choix des classes (explicitement testées) par les gestionnaires des différents systèmes. La transposabilité pour ces systèmes indique que les choix effectués par leurs

responsables des tests s'expliquent par un ou plusieurs critères (reliés aux attributs du code) capturés par les métriques Qi, CBO, LOC et WMC.

- Les responsables des tests des systèmes {DBU et JODA} ont effectué des choix prédictibles à partir de certaines données d'apprentissage uniquement (DBU, LUCENE et POI). Leurs critères de choix ne semblent que partiellement capturés par les métriques.

- Finalement, le groupe formé des systèmes {ANT, IO, MATH} a des résultats difficiles à prédire à partir de n'importe quel ensemble d'entrainement. Pour MATH, et IO cela peut s'expliquer par les remarques soulevées précédemment dans les interprétations des lignes. Pour ANT, cela peut s'expliquer par la faible variabilité des métriques (Qi, CBO, LOC et WMC) (cf. section 6.4.1).

L'apprentissage et la validation croisée montrent qu'en se basant uniquement sur les métriques brutes, il est possible de construire des modèles apprenant sur des données logicielles existantes et capables de proposer automatiquement des ensembles tests pour de nouveaux systèmes logiciels, ensembles qui sont à plus de 70 % relativement similaires à ce qu'aurait proposé un responsable de test connaissant le système. L'apprentissage et la validation inter-systèmes suggèrent aussi que les critères de choix des responsables sont relativement identiques (cohérents) et significativement capturés par les métriques Qi, CBO, LOC et WMC. Il serait intéressant d'étendre cette étude à d'autres métriques.

8.6 Risques pour la validité

L'étude que nous avons menée ici a été effectuée sur 10 systèmes *open source* qui totalisent plus d'un demi-million de lignes de code (628K). L'échantillon est large pour permettre d'obtenir des résultats significatifs, mais les techniques de mesure et la démarche utilisée présentent des limitations qui peuvent restreindre la généralisation de certaines conclusions. En effet, nous avons identifié dans ces démarches deux groupes de risques, un pour la validité externe qui peut empêcher la généralisation des résultats et l'autre sur la validité conceptuelle qui peut biaiser les mesures.

Pour la validité externe, le risque est principalement lié au type de données. L'effet du type de données sur les résultats peut dépendre des domaines d'application des

systèmes. En effet, comme décrit à la section 3.1, certains systèmes étudiés sont des bibliothèques d'algorithmes mathématiques (IO), alors que d'autres ont des architectures plus complexes et font appel à beaucoup de concepts propres à la technologie objet comme l'héritage et le polymorphisme (JFC). Ainsi, l'apprentissage qui s'effectue sur certains types de systèmes peut fortement s'ajuster aux données issues du domaine d'application de ces systèmes et ne pas permettre la proposition des bons ensembles tests pour des systèmes de domaines différents. Il serait intéressant, dans ce cadre, d'introduire dans cette analyse des catégories de systèmes selon le domaine d'application pour tenir compte de ce biais. D'un autre côté, l'analyse des données issues des dépôts de systèmes ne nous garantit pas qu'il y'a eu effectivement des critères de choix systématiques des ensembles tests pour tous les systèmes logiciels. Il se peut que pour certains systèmes, certaines classes aient été choisies au hasard. Dans ces cas, l'apprentissage ne peut être généralisable, même pour des logiciels dans le même domaine d'application.

Pour la validité conceptuelle, le risque principal réside dans la technique utilisée en 3.3 pour l'appariement des classes logicielles aux classes tests. Ce risque pour la validité mentionnée dans les sections 5.5 en tant qu'externe, est lié à la validité conceptuelle dans le cadre de la démarche courante. En effet, les classes de test non appariées ainsi que les classes logicielles qui sont testées de manière indirecte (par appels transitifs des méthodes) sont ignorées par notre approche. La modalité TESTED ne donne que les classes testées explicitement et dont les classes tests correspondent au patron utilisé à la section 3.3 pour l'appariement. Il serait possible de réduire ce biais en considérant les ensembles tests formés des classes logicielles explicitement et implicitement testées. Ces ensembles tests implicites peuvent être fournis par les environnements de développement comme Borland Together [88].

8.7 Conclusion

Nous avons analysé 10 systèmes logiciels *open source* pour lesquels des ensembles tests ont été proposés par les responsables des tests. Les critères des choix des classes nous sont inconnus. Nous avons exploré la possibilité d'expliquer ces critères à travers 3

attributs de code source des logiciels ainsi que les indicateurs de qualité, dans 2 expérimentations. Dans un premier temps, nous avons analysé pour chaque système les groupes testés et les groupes non testés à l'aide du test Z de la moyenne. Les résultats révèlent que pour toutes les métriques sauf CBO, les moyennes sont significativement différentes et leurs valeurs sont cohérentes d'un système à l'autre. En effet, les moyennes des métriques logicielles étaient plus faibles (plus fortes pour Qi) dans les groupes non testés que dans les groupes testés, pour 9 systèmes sur 10. Dans un second temps, nous avons effectué une analyse de régression logistique binaire univariée afin de déterminer les liens individuels des métriques logicielles avec les choix des classes à tester. Les résultats montrent des liens significatifs avec les métriques logicielles qui pourraient expliquer les critères de choix des responsables. Nous avons finalement utilisé le réseau bayésien et la régression logistique multivariée pour construire des modèles d'aide à l'orientation des tests pour un système à partir des données (des métriques et des choix des responsables de test) issues d'autres systèmes. Les résultats suggèrent que plus de 70% des classes logicielles orientées ainsi automatiquement correspondent aux choix des responsables des tests.

Les résultats de cette expérimentation suggèrent la viabilité d'une méthode d'orientation des tests basée sur les métriques logicielles et l'information des tests apprise à partir d'autres systèmes. Il serait intéressant de grouper les systèmes selon leurs domaines d'applications et d'inclure d'autres métriques logicielles (comme RFC) pour observer les variations sur les résultats. Par ailleurs, il serait aussi pertinent d'analyser les performances des ensembles tests proposés automatiquement en termes de ciblage des classes fautives. Ce sujet sera, entre autres, l'objet de notre prochaine section.

CHAPITRE 9. MODÈLE DE RISQUE ET ORIENTATION DES TESTS

9.1 Introduction

Dans cette étude, nous proposons un cadre théorique et expérimental qui peut aider à orienter la distribution de l'effort des tests unitaires des classes dans un système orienté objet. L'approche que nous proposons se base sur la théorie du risque de Bernoulli [145], utilisée en gestion de projet et en ingénierie afin d'évaluer objectivement le risque associé aux composants d'un système. Cette théorie analyse le risque associé à un évènement redouté suivant deux angles que sont la probabilité de survenance d'un évènement non souhaité sur un composant et la gravité de cet évènement, une fois survenu, sur l'ensemble du système. Le risque étant le produit de ces deux grandeurs. Gérer le risque revient d'abord à déterminer les seuils de risque acceptables ou non, et ensuite à ramener les composants qui présentent des valeurs supérieures aux seuils dans des zones dites acceptables. C'est ainsi que le modèle propose, pour chaque composant d'un système (que l'on souhaite analyser), de quantifier objectivement la probabilité d'occurrence de l'évènement redouté et la gravité de l'évènement par rapport à l'ensemble du système (et au-delà), pour ensuite le représenter (projection) sur un repère plan dont l'axe horizontal représente les niveaux de probabilités et l'axe vertical représent les niveaux de gravité (figure 20). Ce repère divise le plan en paliers de risque.

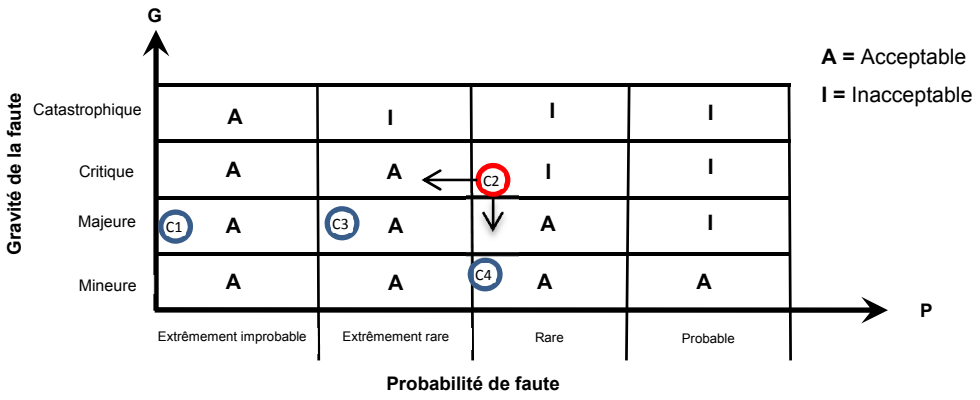


Figure 20: Projections des observations sur le plan P x G du risque.

Un composant se trouvant dans un palier de risque non acceptable peut être ramené dans un palier acceptable en diminuant la gravité ou la probabilité ou les deux dimensions

(en même temps) associées à l'évènement redouté. Les actions qui visent la réduction de la probabilité de survenance de l'évènement redouté forment le groupe des mesures ou d'actions préventives. Ces interventions externes permettent de ramener l'évènement dans une zone acceptable en diminuant sa probabilité de survenance. Les actions qui ciblent la réduction de la gravité en cas de survenance de l'évènement forment le groupe des mesures/actions correctives. Les actions correctives consistent à prendre des mesures qui permettent, entre autre, de contenir l'impact éventuel de l'évènement redouté une fois qu'il se produit afin d'en réduire la gravité.

9.2 Évaluation du risque des classes logicielles

Dans le cadre des logiciels orientés objet, nous nous proposons d'utiliser cette approche, dans un premier temps, pour évaluer objectivement les risques des classes logicielles et, dans un second temps, pour orienter les tests en fonction de cette évaluation. Cette approche pourra éventuellement être utilisée (ultérieurement) pour déterminer les classes candidates à la restructuration, comme action corrective possible. La technique pourrait contribuer à définir des politiques de gestion de risque sur les composants en déterminant le type d'action corrective et/ou préventive (restructuration et/ou test) à apporter aux composants présentant un risque non acceptable (selon notre échelle de risque). Le but n'est pas de mettre en place des techniques de restructuration ou de test, mais plutôt de comprendre les causes du risque non acceptable et d'orienter les actions en conséquence, ce qui rendrait les tests encore plus efficaces. L'identification de l'évènement redouté qui sera sujet d'analyse déterminera les facteurs à mettre en place pour une stratégie d'évaluation objective du risque pour les composants logiciels. En effet, selon la phase du cycle de vie du logiciel, différents évènements peuvent être particulièrement redoutés.

- En phase d'exploitation, un logiciel peut être sujet à des défaillances du fait de plusieurs facteurs externes liés à son utilisation, son environnement, etc., et des facteurs internes liés à ses composants et à leurs interactions. Ces composants peuvent, en effet, contenir des fautes à l'origine d'erreurs pouvant entrainer des défaillances plus ou moins

graves du système. La présence d'une faute dans ces composants peut faire l'objet d'analyse et constituer l'évènement redouté dans le modèle de risque de Bernoulli.

- Lors de la phase de maintenance, les changements peuvent intervenir au niveau des composants suite à la découverte d'une faute, à l'introduction d'une nouvelle fonctionnalité, ou à une adaptation à un nouvel environnement. Certains composants plus exposés sont sujets à des changements, et peuvent impacter l'ensemble (ou une grande partie) du système, compliquant grandement la tâche de maintenance. Les développeurs sont souvent appelés à faire des choix éclairés sur les composants à modifier dans le système. Dans ce contexte, le changement peut faire l'objet d'étude sur sa probabilité et sa gravité, et constituer l'évènement redouté dans le modèle de risque de Bernoulli.

Pour notre étude, nous nous limitons à la présence de faute dans une classe logicielle comme évènement redouté. Ainsi, l'axe horizontal des probabilités (figure 20) dans le modèle de Bernoulli représentera la probabilité qu'une faute survienne dans une classe, et l'axe vertical la gravité de celle-ci sur le système en termes d'impact et de sévérité. En distinguant ces deux dimensions du risque, le responsable pourra agir sur les deux leviers de probabilité et de gravité pour ramener une classe logicielle de la zone de risque inacceptable à la zone de risque acceptable.

9.2.1 La probabilité de présence de faute dans une classe

La prédiction de fautes dans le code source des logiciels a fait l'objet de plusieurs investigations [40]. Plusieurs modèles ont été proposés dans la littérature [43,45,46] et font intervenir différents facteurs parmi lesquels les artéfacts mesurables du logiciel [40,101,131,146,147], l'environnement de développement [35,40], l'expérience des équipes de développement [35], la maturité du processus de développement [35]. Nous nous intéressons aux facteurs liés aux artéfacts mesurables particulièrement aux attributs de code source. Dans ce cadre, différentes études [51,54] montrent que la probabilité de présence de fautes est fortement liée aux métriques logicielles, particulièrement la taille et la complexité cyclomatique [38,53]. La complexité cyclomatique (McCabe [100]) détermine le nombre de chemins possibles (suivant les embranchements des structures de contrôle)

dans une méthode. Le taux de couverture de test des méthodes (exprimé en pourcentage) indique le rapport entre le nombre de chemins couverts par les tests et le nombre total de chemins (la complexité cyclomatique). Ce taux renseigne sur l'effort de test investi pour cette méthode, mais aussi et surtout sur la diminution de la probabilité de faute [45]. On peut raisonnablement supposer que la probabilité de faute P dans une classe C_i est proportionnelle à sa complexité $VG(C_i)$, et à $(1 - t_i)$ où t_i est son taux de couverture de test.

$$P(C_i) = k * (1 - t_i) * VG(C_i)$$

Cette approximation naïve de la probabilité de faute peut être utilisée pour évaluer la probabilité de présence de fautes dans un composant. Dans le cadre des tests unitaires, par exemple, le responsable des tests peut prendre des mesures préventives en augmentant la couverture des tests pour réduire la probabilité de faute dans cette classe et par conséquent le risque qui lui est associé dans cette dimension.

Un autre moyen de réduire la probabilité de présence de fautes consiste à diminuer la taille des classes (les fautes étant liées à la taille des classes aussi [45]), et leur complexité, en restructurant (remaniant) ces composants en deux ou plusieurs classes plus petites ce qui permet au passage d'améliorer (souvent) la cohésion. Le responsable peut contrôler le paramètre de la taille (par exemple) des classes pour réduire la probabilité de faute. Il s'agit là d'un autre levier préventif permettant de diminuer la probabilité de survenance de faute dans une classe.

9.2.2 La gravité de faute d'une classe

La gravité d'une faute est difficile à estimer, particulièrement dans le cadre du logiciel. En effet, différents facteurs externes (contexte d'utilisation, criticité du domaine) peuvent fortement impacter son évaluation. Ces facteurs sont difficilement quantifiables [83-85]. Nos investigations utilisent uniquement les artéfacts mesurables du code source logiciel. Nous aborderons, donc, la gravité d'une faute dans un système logiciel en termes d'impact sur le système logiciel uniquement, et plus précisément sur sa potentielle capacité

de propagation à travers les composants du logiciel (sévérité). Les classes fortement couplées présentent un grand potentiel d'impact sur le système. En effet, la présence de fautes dans ces classes peut conduire à des erreurs pouvant potentiellement se propager (à cause du couplage) dans le reste du logiciel entrainant la défaillance du système [148]. L'impact de telles fautes est potentiellement grave.

Restructurer les couplages entrants et sortants d'une classe (entre autres) permet de diminuer sa criticité, et donc la dépendance du système (vis-à-vis de ce composant et les répercussions d'une potentielle erreur due aux fautes s'y trouvant). Ceci peut circonscrire l'impact des fautes de cette classe sur le système. À ce niveau, nous pensons que les patrons architecturaux pourraient éventuellement jouer un rôle important.

Implémenter des contrats, devant les entrées-sorties d'une classe logicielle et mettre en place des gestionnaires d'exception autour de certaines parties du code, permettrait de signaler assez tôt les erreurs dues aux fautes, ou d'orienter leur flux (leur propagation) vers des sorties contrôlées afin de réduire l'impact des fautes. Ces mesures correctives permettraient de contenir l'impact d'une faute, et d'en réduire la gravité.

9.2.3 Familles de métriques logicielles dans le modèle de risque

En nous basant sur l'analyse effectuée dans les 2 sous-sections (9.2.1 et 9.2.2), nous pouvons regrouper les métriques orientées objet en 2 catégories selon les 2 dimensions du risque issues de l'approche de Bernoulli (figure 21). Une famille (P) de métriques qui influencent la probabilité de présence de fautes. Il s'agit, en général, de métriques intrinsèques aux classes qui mesurent des caractéristiques internes. Dans ce groupe, nous retrouvons, entre autres, la taille (LOC) et la complexité cyclomatique (WMC). Une famille (G) de métriques qui influencent la gravité d'une faute en termes d'impact. Il s'agit, en général, des familles de métriques qui capturent les interactions entre les différentes classes du système. On retrouve, entre autres, dans cette famille les métriques de couplage sortant (FANOUT), les métriques de couplage entrant (FANIN) et les métriques qui combinent ces deux types de couplage (CBO).

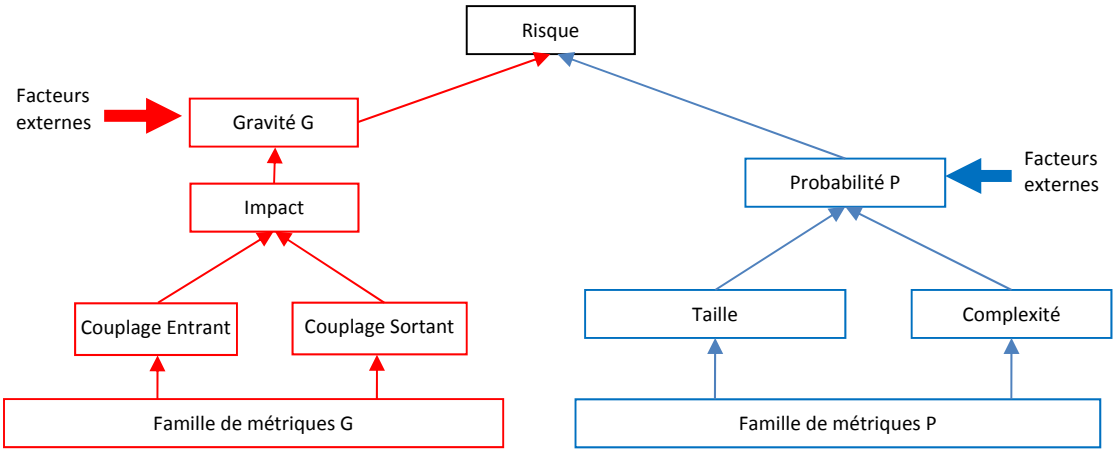


Figure 21: Famille de métriques de code source et facteurs de risque.

Les métriques synthétiques comme les Qi interviennent dans les deux dimensions du risque en capturant à la fois la complexité et le couplage sortant de manière indirecte.

Pour évaluer le risque d'une classe selon notre approche, nous allons considérer des combinaisons d'au moins 2 métriques issues nécessairement de ces deux dimensions complémentaires du risque. Ce choix abonde dans le sens des conclusions de Rosenberg [85] qui suggère qu'un attribut logiciel seul ne devrait jamais être utilisé pour évaluer le risque. Nous utiliserons de préférence des métriques indépendantes pour maximiser l'information capturée et faciliter l'interprétation des projections planes. Pour les évaluations de risque utilisant les Qi dans l'axe des probabilités (figure 20), nous couplerons aux indicateurs de qualité (au moins) une métrique de couplage entrant (qu'ils ne capturent pas directement ou indirectement de par leur formulation).

9.2.4 Niveaux de risque des classes logicielles

En considérant la présence de fautes comme évènement redouté, il est possible de construire avec les métriques de code source une approche simple d'analyse du risque pour les systèmes orientés objet en 4 étapes:

- (1) Choisir une métrique ou une combinaison de plusieurs métriques reliées à la probabilité de faute (par exemple la complexité cyclomatique ou la taille), et une métrique ou une combinaison de plusieurs métriques reliées à la propagation de fautes (par exemple FANIN/FANOUT/CBO).

- (2) Évaluer ces métriques pour toutes les classes du système. Si plusieurs métriques sont choisies pour chaque dimension (probabilité et gravité), les combiner selon les modèles de prédiction de faute et de propagation issus des études (exemple: [45,51]).
- (3) Projeter les classes logicielles en fonction de leur évaluation dans un repère cartésien plan. Dans ce repère, les abscisses représenteront les probabilités de fautes (pondérées par la couverture de tests unitaires), et les ordonnées la gravité de faute en termes d'impact potentiel sur le système.
- (4) La famille de droites (lignes de niveau) $y_{k,a}(x) = k - ax$ ou k et a sont des réels positifs, constituent les différentes échelles de niveaux de qualité possible que l'on peut exiger des composants du système (figure 22). En analysant une ligne de niveau $y_{k,a}(x)$, on note que k (l'ordonnée à l'origine) est l'impact maximum acceptable pour un composant et $\frac{k}{a}$ est la probabilité de présence de faute maximale acceptable dans un composant. Les classes logicielles placées au-dessus de la droite (après leur évaluation) ont un risque inacceptable pour le système. Plus *k* et *-a* sont grands plus les risques acceptables sont grands (et plus les exigences de qualité sont faibles).

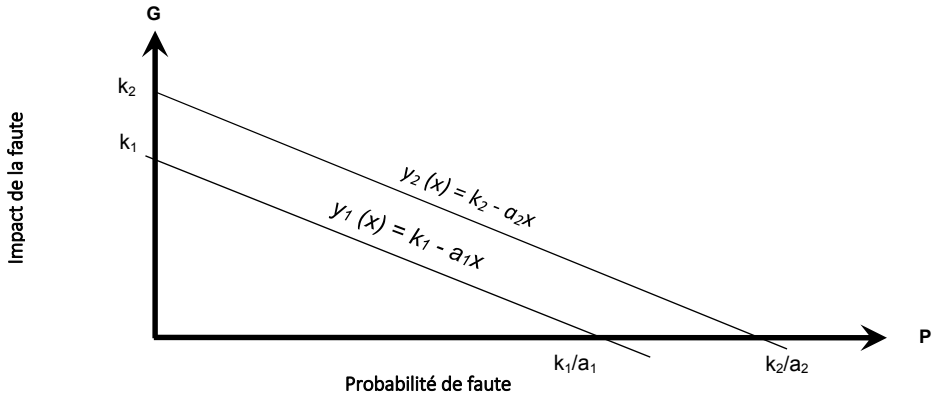


Figure 22: Lignes de niveaux de risque.

9.3 Métriques et logiciels choisis

Nous avons besoin pour les protocoles expérimentaux de coupler des métriques issues de la famille P à celles issues de la famille G. Se basant sur l'existence de liens démontrés ([46,149,150]) entre la complexité et la présence de fautes, nous avons choisi la complexité cyclomatique (WMC) pour représenter la famille P des facteurs de probabilité de présence de fautes dans un composant, à laquelle nous avons joint la métrique Qi qui capture WMC. Pour représenter la famille G (la gravité), nous faisons appel aux métriques de couplage CBO et FANIN. Ces métriques sont décrites dans la section 3.2.1. Cependant, des analyses de corrélation montrent une relation significative entre CBO et WMC [119,121], et entre CBO et Qi [24], ce qui n'est pas le cas entre FANIN et WMC ou entre FANIN et Qi (de par la formulation des Qi). La métrique FANIN est surtout utilisée, d'après [98], pour indiquer les modules critiques qui nécessitent une restructuration ou un effort de test particulier. Nous avons préféré choisir un couple de métriques indépendantes pour diminuer l'effet de recoupement de l'information qu'elles capturent. Ainsi, la valeur de la métrique FANIN d'une classe sera son ordonnée dans nos différentes projections. En résumé, nous avons choisi les couples {WMC x FANIN}, {Qi x FANIN} pour projeter les observations sur le plan.

Par ailleurs, nous avons sélectionné les données de 4 systèmes logiciels *open source* de tailles différentes et de domaines d'application différents, parmi ceux décrits dans la section 3.1. Il s'agit de JFC, JODA, LOG4J et POI. Plusieurs classes de ces systèmes ont été testées explicitement de manière unitaire.

Les statistiques descriptives des 4 systèmes et métriques sont données par les tables 44 et 45. Pour la table 44, les colonnes #CL, %CLT, LOC et %LOCT représentent respectivement le nombre de classes logicielles, le pourcentage de classes explicitement testées, le nombre de lignes de code et le pourcentage de lignes de code testées que représentent les classes explicitement testées. Pour tous les systèmes, les classes purement abstraites et les interfaces ont été éliminées du décompte. Dans la table 45, la

variable TESTED est binaire et indique qu'une classe est explicitement testée quand elle est à 1 et 0 sinon.

Table 44: Statistiques descriptives des systèmes.

	#CL	%CLT	#LOC	%LOCT
JFC	411	55.60%	67 481	78.73%
JODA	201	37.81%	31 339	56.24%
LOG4J	231	19.05%	20 149	38.27%
POI	1382	28.00%	130 184	44.86%

Table 45: Statistiques descriptives des métriques.

	TESTED	Qi	FANIN	CBO	WMC
Obs.	2225	2225	2225	2225	2225
Min	0	0	0	0	0
Max	1	1	189	168	470
Somme	736	1467.71	11 398	17 815	47 201
μ	0.33	0.660	5.12	8.01	21.21
σ	S|O	0.35	10.35	11.25	33.17

POI est le plus large des 4 systèmes en termes de lignes de code et de nombre de classes. Le système LOG4J contenant 20K lignes de code réparties dans 231 classes logicielles, vient en 3ème position en termes de nombre de classes (avant JFC 411 classes), mais en 4ème position en termes de lignes de code après JODA (31K). LOG4J a le taux de classes explicitement testées le plus bas 19.05% représentant 38.27% des lignes de code. JFC compte le plus de classes explicitement testées (55.60%). La table 45 quant à elle, donne le min, le max, la somme, la moyenne (μ) et l'écart-type (σ) des métriques logicielles. Elle montre globalement que sur 2225 classes logicielles considérées, seules 33% (soit 736 classes) ont été explicitement testées (moyenne de TESTED).

9.4 Projections

Nous avons réalisé 3 groupes de protocoles expérimentaux pour explorer l'utilisation de notre modèle de risque basé sur les attributs logiciels dans l'orientation de l'effort global des tests. L'objectif général est d'évaluer l'utilisabilité du modèle dans le cadre des tests unitaires. Nous avons, dans un premier temps, observé les tendances des données des classes logicielles sous l'angle de la projection plane du modèle de risque exposé dans le chapitre précédent. Par la suite, nous avons utilisé les algorithmes d'apprentissage automatique pour évaluer la capacité de prédictions des ensembles tests proposés par les responsables de tests. Nous avons finalement regroupé de manière non supervisée les

classes les plus à risque pour étudier de manière empirique le taux et les sévérités des fautes qu'elles peuvent contenir.

9.4.1 Projections par valeurs des observations et interprétations

L'objectif de cette première étude est de visualiser les informations des tests unitaires explicites effectués dans les systèmes en utilisant notre modèle de projection. Nous avons construit un modèle naïf couplant une métrique de la famille P à une métrique de la famille G. La probabilité est directement représentée par une métrique liée à la probabilité de présence de fautes (axe des abscisses), tandis que la gravité est représentée par une métrique liée à la propagation des erreurs dues aux fautes (axe des ordonnées). Nous avons couplé aux observations l'information sur les tests explicites des classes (modalités 1 et 0). L'objectif est d'étudier d'éventuels patrons de choix des classes explicitement testées sous l'angle du risque. Nous voulons, en effet, vérifier si avec leurs connaissances des systèmes, les responsables des différents projets proposent des ensembles tests en fonction de la probabilité et de l'impact (potentiel sur le système) des fautes que peuvent contenir ces classes (cette évaluation peut être formelle ou intuitive). Il serait possible d'utiliser les lignes de niveaux décrits précédemment pour orienter de futurs tests unitaires.

Après évaluation des métriques Qi, WMC, et FANIN, chaque classe logicielle est placée dans le repère plan selon les valeurs de ses métriques Qi ou WMC sur l'axe des abscisses et FANIN sur l'axe des ordonnées. Les données des tests unitaires y sont superposées comme suit: les classes logicielles explicitement testées apparaissent sous forme de triangles (bleus) tandis que celles qui ne sont pas explicitement testées apparaissent sous forme de cercles (rouges). Nous avons par ailleurs, effectué les mêmes projections graphiques en utilisant les rangs des métriques (à la place de leurs valeurs directes) afin de diminuer l'effet de la variance, ce qui donne une meilleure visualisation des patrons de projection. Le modèle peut se généraliser avec des combinaisons appropriées de métriques sur chaque axe, combinaisons qui peuvent être issues des modèles de prédiction linéaires, logistiques ou d'apprentissage automatique des

probabilités et d'impact de fautes à partir de métriques logicielles. Les figures 23-26 montrent les résultats de la projection des données superposées à l'information sur les classes explicitement testées pour chaque système selon les valeurs directes des métriques (Qi, WMC, et FANIN) et les rangs des valeurs des métriques (R-Qi, R-WMC, et R-FANIN) sur les abscisses et les ordonnées.

La projection basée sur les rangs présente les données sur la même échelle et est moins sensible à la variance, ce qui donne une meilleure visualisation de l'information sur le repère. Celle basée sur les valeurs directes affiche les données à leur échelle réelle, ce qui nuit à la lisibilité de la projection lorsque la variabilité est importante.

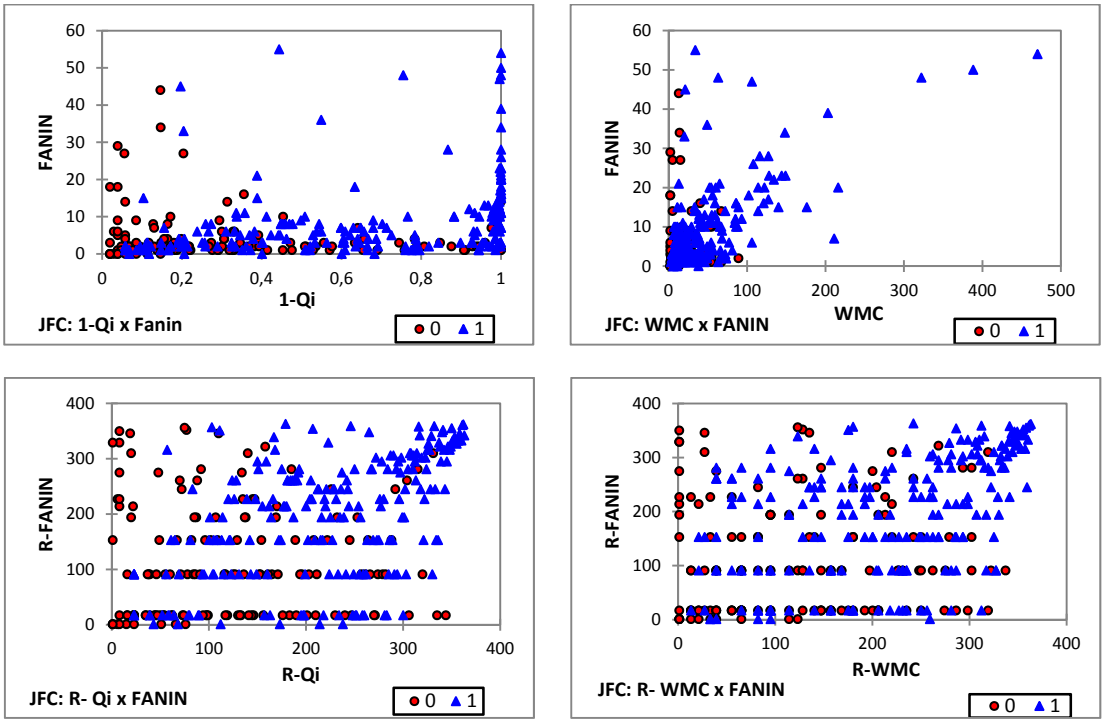


Figure 23: Projections planes des observations de JFC.

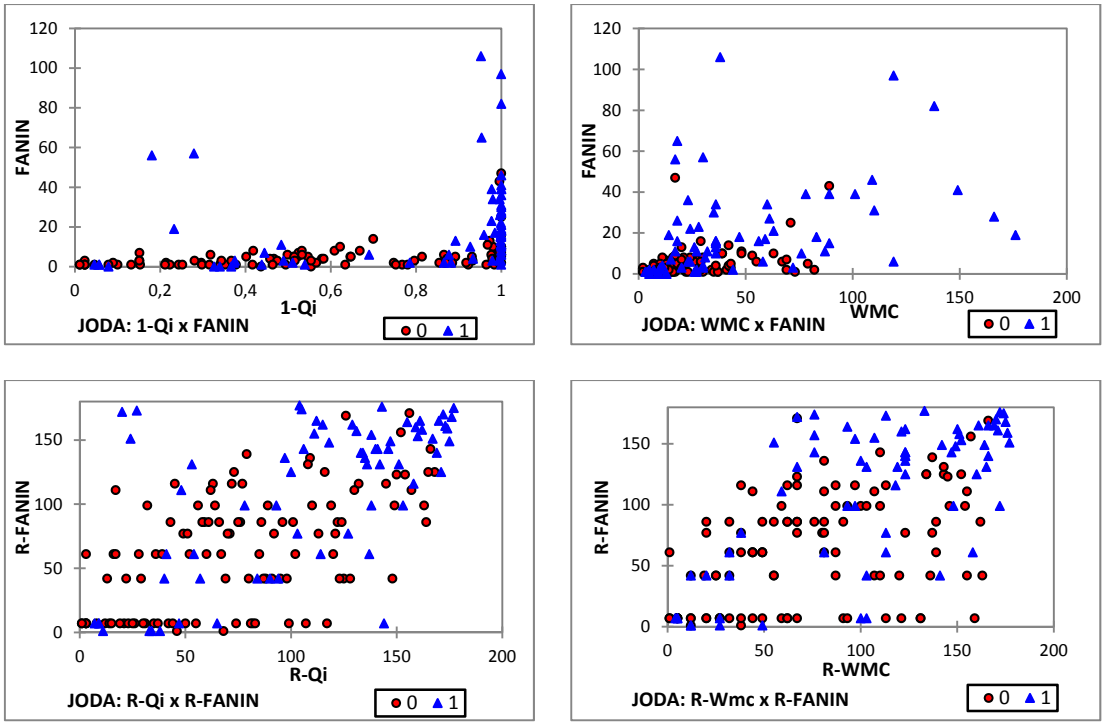


Figure 24: Projections planes des observations de JODA.

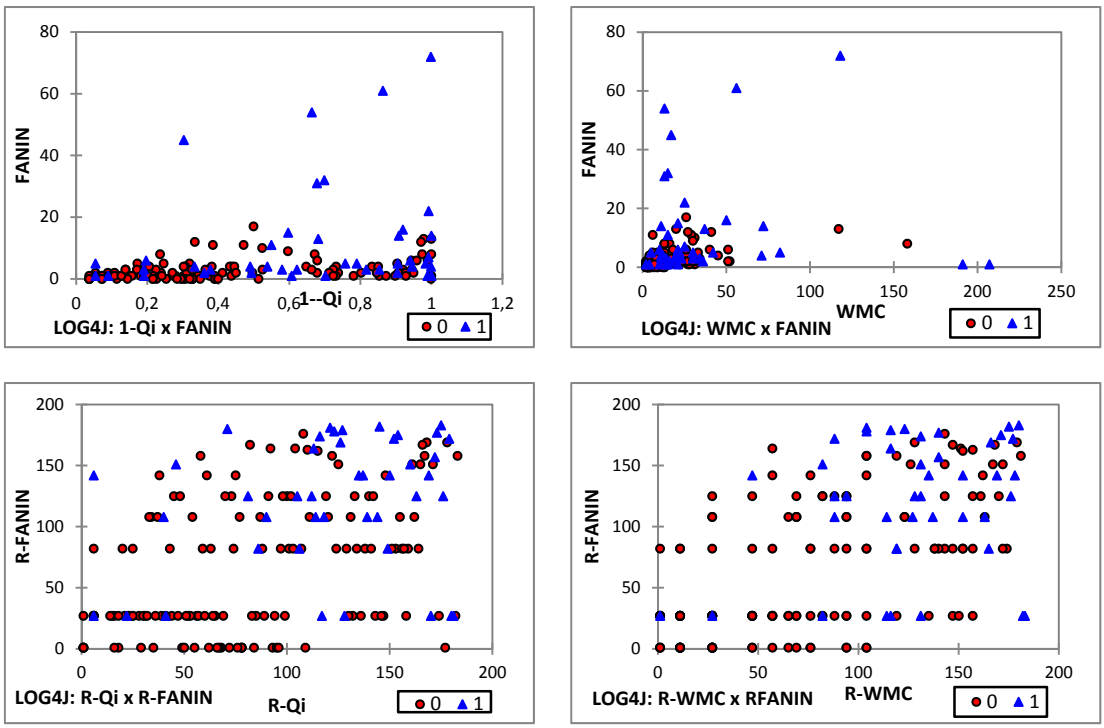


Figure 25: Projections planes des observations de LOG4J.

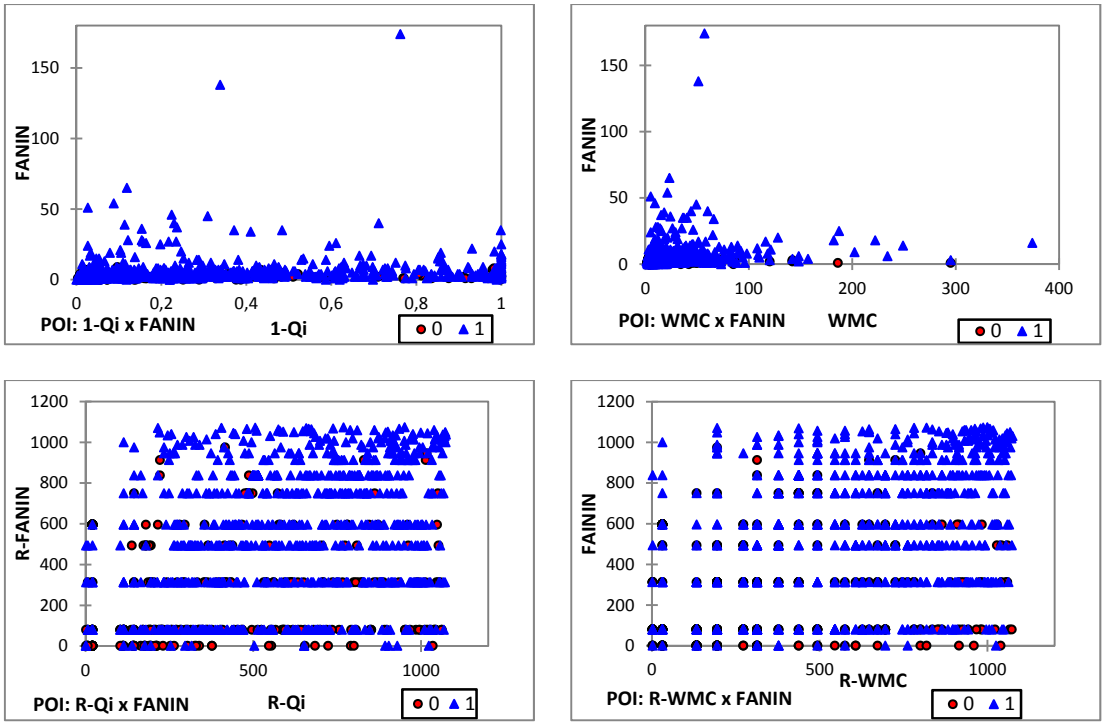


Figure 26: Projections planes des observations de POI.

Nous constatons que dans les projections basées sur les rangs, les classes logicielles placées en haut à droite ont été généralement explicitement testées. Cette tendance traduit l'existence de lien entre les classes explicitement testées et les couples de métriques {FANIN x Qi} et {FANIN x WMC}. En effet, nous constatons que les classes complexes (fort WMC, faible Qi) et très sollicitées (par plusieurs autres classes du système (forte valeur de FANIN)) ont été pour la plupart explicitement testées par les responsables des tests.

Notons que les classes à fort potentiel d'impact (en termes de propagation) et faible probabilité de fautes ne sont pas testées. Ces classes ont le profil de classes utilitaires contenant des fonctionnalités simples et sont invoquées d'un peu partout dans le code.

Nous remarquons également que les classes complexes, même avec un impact faible, sont quand même testées. Deux raisons (entre autres) peuvent expliquer ce choix: (1) La complexité cyclomatique est un fait visible sur le code d'une classe ce qui induit des choix presque systématiques de ces classes pour des tests unitaires explicites même si leur impact potentiel n'est pas relativement important, alors que la propagation implique une

vision architecturale des liens de dépendance entre les classes, ce qui nécessite une analyse plus poussée du système, analyse qui n'est généralement pas faite de manière systématique lors des tests. (2) La métrique FANIN capture juste le couplage entrant de premier niveau, ce qui veut dire que le FANIN d'une classe donnée peut être faible alors que son potentiel d'impact est fort sur le système. C'est le cas, par exemple, lorsque la classe est utilisée par une classe intermédiaire à fort FANIN (fort potentiel impact). Par ailleurs. Le système POI semble projeter un profil singulier. En effet, il est particulièrement difficile de lire les données avec cette approche lorsque la taille du système est très grande (POI étant le plus grand système en termes de nombre de classes et de lignes de code). Ce constat va motiver notre seconde approche de visualisation.

9.4.2 Projections par catégories des observations et interprétations

Nous voulons, dans cette sous-section, confirmer et quantifier les tendances observées dans les projections directes précédentes, particulièrement dans le cas des systèmes de grande taille ou de faible variance. En effet, dans cette approche, nous avons partitionné (en utilisant le partitionnement uniforme décrit à la section 3.4.2.2) les observations selon les métriques en 5 clusters. Le partitionnement uniforme regroupe les observations en clusters (pour chaque attribut considéré: FANIN, WMC, Qi) en minimisant la variance intra-cluster et en maximisant la variance inter-clusters. Nous avons choisi 5 clusters pour correspondre aux différents niveaux symboliques de risques: TF (très faible), F (faible), M (moyen), E (élevé) et TE (très élevé). Les observations sont ensuite projetées dans le repère (qui a désormais 5x5 valeurs constituant les paliers de risque).

Après analyse des moyennes des clusters obtenus par le partitionnement uniforme, nous les avons associées aux différents niveaux par ordre croissant de la moyenne. Pour les métriques FANIN et WMC, le cluster à la moyenne la plus grande correspond au niveau TE et la plus petite au niveau TF, en passant par les niveaux intermédiaires dans l'ordre décroissant. Pour la métrique Qi, le cluster ayant la moyenne la plus grande correspond au niveau TF et la plus petite au TE, en passant par les niveaux intermédiaires dans l'ordre

croissant. Cette inversion est nécessaire pour le cas de la métrique Qi à cause de sa formulation (chapitre 4).

Dans chacune des 25 cases obtenues (paliers de risque), nous avons calculé le taux de test explicite. Ce taux est donné par le rapport entre le nombre de classes explicitement testées dans ce palier et le nombre total de classes dans ce palier. Les croix (X) indiquent l'absence de classes logicielles dans ce palier. Les tables 46-49 présentent les résultats pour chaque système selon la valeur directe des métriques (FANIN, WMC, QI) et le rang de la valeur des métriques (R-FANIN, R-WMC, R-Qi).

Table 46: Taux de classes testées selon les paliers de risque (JFC).

R-FANIN						
TE	0.0 %	50.0 %	100.0 %	83.3 %	97.7 %	
E	55.6 %	76.9 %	76.2 %	86.7 %	85.7 %	
M	37.5 %	35.7 %	64.3 %	77.8 %	85.7 %	
F	37.5 %	41.7 %	50.0 %	56.3 %	50.0 %	
TF	17.6 %	60.0 %	33.3 %	66.7 %	33.3 %	
	TF	F	M	E	TE	R-WMC

R-FANIN						
TE	97.7 %	50.0 %	100.0 %	0.0 %	83.3 %	
E	85.7 %	76.9 %	76.2 %	55.6 %	86.7 %	
M	85.7 %	35.7 %	64.3 %	37.5 %	77.8 %	
F	50.0 %	41.7 %	50.0 %	37.5 %	56.3 %	
TF	33.3 %	60.0 %	33.3 %	17.6 %	66.7 %	
	TF	F	M	E	TE	R-QI

FANIN						
TE	50.0 %	100.0 %	100.0 %	x	100.0 %	
E	20.0 %	100.0 %	100.0 %	100.0 %	x	
M	42.9 %	87.5 %	100.0 %	100.0 %	x	
F	69.6 %	88.5 %	100.0 %	100.0 %	x	
TF	41.6 %	69.0 %	50.0 %	x	x	
	TF	F	M	E	TE	WMC

FANIN						
TE	50.0 %	X	100.0 %	100.0 %	100.0 %	
E	20.0 %	X	X	100.0%	100.0 %	
M	25.0 %	50.0 %	X	100.0 %	96.7 %	
F	11.1 %	73.3 %	93.3 %	90.0 %	90.0 %	
TF	36.4 %	43.6 %	60.0 %	71.9 %	61.3 %	
	TF	F	M	E	TE	Qi

Table 47: Taux de classes testées selon les paliers de risque (JODA).

R-FANIN						
TE	X	66.7%	100.0 %	100.0 %	88.9 %	
E	X	28.6 %	42.9 %	57.1 %	44.4 %	
M	0.0 %	10.0 %	18.2 %	20.0 %	40.0 %	
F	36.4 %	0.0 %	40.0 %	16.7 %	50.0 %	
TF	30.4 %	11.1 %	50.0 %	0.0 %	0.0 %	
	TF	F	M	E	TE	R-WMC

R-FANIN						
TE	100.00	X	100.00	87.50 %	88.24 %	
E	0.00 %	40.00 %	42.86 %	54.55 %	46.15 %	
M	0.00 %	0.00 %	30.00 %	25.00 %	25.00 %	
F	0.00 %	44.44 %	27.27 %	33.33 %	0.00 %	
TF	26.09 %	30.00 %	0.00 %	0.00 %	100.00	
	TF	F	M	E	TE	R-Qi

FANIN						
TE	X	100.0%	X	100.0 %	100.0 %	
E	100.0 %	100.0 %	X	X	X	
M	50.0 %	100.0 %	80.0 %	100.0 %	100.0 %	
F	80.0 %	66.7 %	77.8 %	X	100.0 %	
TF	18.4 %	30.8 %	20.0 %	100.0 %	X	
	TF	F	M	E	TE	WMC

FANIN						
TE	X	X	100.00	X	X	
E	100.00	100.00	X	X	100.00	
M	X	X	X	X	85.71 %	
F	X	100.00	50.00 %	50.00 %	75.00 %	
TF	21.05 %	21.74 %	16.67 %	27.78 %	23.53 %	
	TF	F	M	E	TE	QI

Table 48: Taux de classes testées selon les paliers de risque (LOG4J).

R-FANIN						
TE	X	25.0%	57.1 %	57.1 %	47.1 %	
E	0.0 %	0.0 %	30.0 %	62.5 %	42.9 %	
M	0.0 %	0.0 %	0.0 %	25.0 %	16.7 %	
F	8.3 %	0.0 %	14.3 %	50.0 %	50.0 %	
TF	0.0 %	0.0 %	0.0 %	X	X	
	TF	F	M	E	TE	R-WMC

R-FANIN						
TE	100.0 %	40.0 %	0.0 %	69.2 %	50.0 %	
E	0.0 %	16.7 %	30.0 %	41.7 %	50.0 %	
M	0.0 %	0.0 %	25.0 %	0.0 %	14.3 %	
F	9.5 %	6.7 %	0.0 %	28.6 %	42.9 %	
TF	0.0 %	0.0 %	0.0 %	x	0.0 %	
	TF	F	M	E	TE	R-Qi

FANIN						
TE	100.0 %	X	100.0 %	100.0 %	X	
E	100.0 %	X	X	X	X	
M	66.7 %	33.3 %	100.0 %	0.0 %	X	
F	21.1 %	46.7 %	66.7 %	0.0 %	X	
TF	9.0 %	30.0 %	0.0 %	X	100.0 %	
	TF	F	M	E	TE	WMC

FANIN						
TE	X	X	X	100.0%	100.0 %	
E	X	100.0 %	X	100.0 %	X	
M	X	0.0 %	20.0 %	100.0 %	57.1 %	
F	100.0 %	20.0 %	33.3 %	28.6 %	42.9 %	
TF	5.7 %	5.9 %	15.0 %	30.8 %	27.3 %	
	TF	F	M	E	TE	Qi

Table 49: Taux de classes testées selon les paliers de risques (POI).

R-FANIN						
TE	10.0 %	14.3 %	37.5 %	59.4 %	67.6 %	
E	10.0 %	36.1 %	36.4 %	60.0 %	61.8 %	
M	3.0 %	23.7 %	34.1 %	63.0 %	56.3 %	
F	9.1 %	19.6 %	53.8 %	34.4 %	36.4 %	
TF	6.4 %	15.2 %	20.5 %	24.3 %	36.0 %	
	TF	F	M	E	TE	R-WMC

R-FANIN						
TE	0.0 %	13.0 %	48.1 %	53.7 %	66.7 %	
E	0.0 %	29.4 %	39.2 %	55.0 %	65.5 %	
M	3.1 %	21.4 %	36.9 %	54.8 %	58.1 %	
F	0.0 %	25.5 %	24.3 %	37.1 %	35.6 %	
TF	6.2 %	13.0 %	19.0 %	25.0 %	24.3 %	
	TF	F	M	E	TE	R-Qi

FANIN						
TE	x	x	x	x	x	
E	x	x	x	x	x	
M	x	x	100.0 %	x	x	
F	66.7 %	66.7 %	40.0 %	100.0 %	x	
TF	19.0 %	51.6 %	57.5 %	50.0 %	66.7 %	
	TF	F	M	E	TE	WMC

FANIN						
TE	X	X	X	X	X	
E	X	X	X	X	X	
M	X	X	100.0 %	100.0 %	X	
F	0.0 %	78.6 %	75.0 %	0.0 %	100.0 %	
TF	12.9 %	36.0 %	43.8 %	53.6 %	50.4 %	
	TF	F	M	E	TE	Qi

Pour tous les systèmes partitionnés et projetés, nous constatons généralement que les classes très sollicitées et peu complexes (de TFxTF à TFxTE) ont un faible taux de test explicite. Le risque associé à ces classes reste marginal, malgré un impact potentiel pouvant être fort sur l'ensemble du système, pour les classes du palier TFxTE. Ce qui peut s'expliquer par le fait que la faible complexité assure la quasi-absence de faute, d'où la confiance intuitive des responsables des tests vis-à-vis de ces classes malgré un fort potentiel d'impact de fautes dans les classes de TFxTE. Ces classes ont le profil de classes convoyeuses d'information (beans) ou des classes utilitaires simples. Les développeurs ont plutôt explicitement testé les classes complexes et de préférence très sollicitées (de MxM à TExTE). En effet, dans cette catégorie, le risque est très élevé, dû à la probabilité élevée de

faute et à leur important potentiel d'impact sur le système. Cette catégorie de classes est la cible première des testeurs. Les catégories de classes complexes et peu sollicitées (TFxTF à TExTF) ont été partiellement testées (1 % à 50 %). Cela peut s'expliquer par le fait qu'au premier coup d'œil, à travers le code, la complexité est la face visible du risque associé à la présence de faute dans une classe.

Certains paliers de risque ne contiennent pas d'observations, particulièrement dans les projections impliquant les valeurs directes des métriques. Ainsi, à cause d'une faible variance des valeurs de FANIN, le partitionnement uniforme n'a pu déterminer automatiquement que 3 classes de FANIN dans le cas de POI.

L'utilisation des rangs des métriques permet de relativiser les risques des classes les uns par rapport aux autres et de ramener les variances des métriques à des valeurs comparables. Cette technique reflète mieux la vision du développeur d'un système qui juge les (risques des) classes les unes par rapport aux autres pour décider de la nécessité de les tester explicitement ou pas. L'utilisation directe des valeurs numériques des métriques reflète la vision des développeurs d'expérience, ayant une vue du système, et le comparant à d'autres systèmes. Les valeurs directes sont fortement soumises au biais de la variance et donc des observations extrêmes.

9.5 Apprentissage basé sur le risque et orientation des tests

Dans cette section, nous abordons la problématique de l'orientation de l'effort global de test basée sur l'analyse du risque. L'objectif est de suggérer, pour un système donné, un ensemble test (ensemble de classes logicielles pour lesquelles il faudra développer des classes tests unitaires) avec les techniques d'apprentissage supervisé. Suite aux observations issues des techniques de visualisation précédentes, nous utilisons dans cette section des méthodes d'intelligence artificielle afin de suggérer de manière automatique des ensembles tests parmi les classes logicielles d'un système (orientation). L'approche d'orientation automatique sera basée sur les données d'analyse du risque obtenues à partir d'autres systèmes logiciels. Nous croyons, en effet, que l'utilisation de métriques

logicielles adéquates pourrait permettre, par apprentissage automatique, de proposer des ensembles tests adéquats pour un nouveau système.

Pour un système logiciel donné dont on connait l'ensemble test proposé par les responsables des tests, nous calculons les métriques Qi, WMC, FANIN et CBO pour chaque classe. Nous assignons aussi à chaque classe logicielle une variable binaire TESTED prenant valeur dans l'ensemble {1, 0} selon que le responsable de test l'ait explicitement testée (ou choisie pour être explicitement testée) ou pas. Nous avons ensuite utilisé 3 algorithmes d'apprentissage automatique, adéquats pour le type de données de notre problématique. Ils exigent peu ou pas de conditions particulières sur nos données d'observations. Ces algorithmes sont, par ailleurs, largement utilisés et permettent de construire de bons modèles de prédiction. Il s'agit de :

- KNN, présenté à la section 3.4.2.2. Son choix s'explique par son nombre réduit de paramètres (1 seul) mais aussi et surtout pour son utilisation répandue dans la reconnaissance de patrons. Dans nos expérimentations, nous avons exploré les modèles à 1, 3 et 5 noyaux. Les meilleurs taux d'erreur sur les données d'entrainement ont été obtenus quand ce paramètre est fixé à 3, paramètre que nous avons gardé par la suite.
- C4.5 est aussi présenté à la section 3.4.2.2. Le choix de cet algorithme est motivé par des besoins de comparaison avec le KNN, mais aussi par sa capacité de manipulation des données discrètes. Particulièrement, si l'on souhaite ultérieurement considérer des valeurs qualitatives des niveaux de probabilité et de gravité des fautes. Le paramètre de taille minimale des feuilles est fixé à 5.
- LOG est la régression logistique binaire présentée dans la section 3.4.1.3. Elle est idéale pour la prédiction de variables à deux modalités comme dans notre cas.

Nous avons, par ailleurs, utilisé les ensembles de classificateurs BOOSTING et BAGGING, décrits en 3.4.2.3, pour tenter d'améliorer leurs résultats. Les performances de ces algorithmes ont été évaluées grâce à différentes approches de validation (FITTING, 10-FV, 10x70/30 et le LOOV) que nous avons présentées à la section 3.4.2.4. L'objectif est de vérifier si ces algorithmes sont suffisamment performants pour construire des prédicteurs

fiables sur les données d'un système avant d'envisager les validations entre des systèmes différents.

Nous avons choisi, pour cette expérimentation, les couples de métriques du modèle de risque (construit plus haut) représentant les axes de probabilité (abscisses) et de gravité (ordonnées) des fautes comme suit: {WMC x FANIN}, {Qi x FANIN}, {WMC x CBO}, {Qi x CBO}. Nous avons, par ailleurs, sélectionné les systèmes JFC et POI pour mener cette expérimentation. JFC a été choisi, car la section 8.5 a montré qu'il s'agissait d'un bon ensemble d'entrainement et de test avec une bonne répartition des observations (50%-50%). POI a été choisi, car il est le plus grand système de par son nombre de classes et de lignes de code, mais aussi à cause de sa distribution (de la variable TESTED) très courante d'environ 30%-70%.

Nous avons résumé dans les tables 50 et 51, pour les deux systèmes, les performances des 3 algorithmes d'apprentissage automatique ainsi que leurs versions dopées. Dans ces tables, les colonnes FITTING donnent les taux d'erreur lorsque l'ensemble de test est le même que l'ensemble d'entrainement. Le BOOSTING et BAGGING sont répliqués 50 fois (50 votants) pour chacune des 3 techniques d'apprentissage et les colonnes correspondantes contiennent les taux d'erreur globaux. Les colonnes 10x70/30 représentent les erreurs moyennes de 10 itérations avec une division aléatoire du jeu de données initial en ensembles d'entrainement (70%) et en ensembles de test (30 %). Il s'agit d'une méthode de validation applicable avant et après les ensembles de classificateurs, permettant donc d'évaluer l'amélioration après le BOOSTING et BAGGING. Les colonnes 10xFV contiennent les erreurs moyennes des 10-Fold cross-validation. Et enfin, les colonnes LOOV contiennent les erreurs globales issues de la technique de validation Leave-One-Out-Validation.

Table 50: Taux d'erreur des algorithmes d'apprentissage pour JFC.

		Sans BOSTING / BAGGING				BOOSTING		BAGGING	
KNN	{PxG}	FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.1371	0.2302	0.2347	0.2218	0.0242	0.2638	0.1190	0.2275
	{WMC x FANIN}	0.1774	0.3081	0.3020	0.2863	0.1593	0.3000	0.1552	0.2738
	{Qi x CBO}	0.1472	0.2490	0.2490	0.2500	0.1331	0.2617	0.1190	0.2389
	{WMC x CBO}	0.1996	0.3255	0.3224	0.3306	0.1774	0.3416	0.1875	0.3356
C4.5		FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.127	0.2369	0.2184	0.1996	0.0202	0.2020	0.0968	0.2228
	{WMC x FANIN}	0.1613	0.2832	0.2816	0.2823	0.1371	0.2779	0.1532	0.2638
	{Qi x CBO}	0.1532	0.2423	0.2408	0.2563	0.1048	0.2107	0.1331	0.2148
	{WMC x CBO}	0.1956	0.3242	0.3469	0.3831	0.1069	0.3383	0.1714	0.2993
LOG		FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.2379	0.2517	0.2510	0.2419	0.2560	0.2470	0.2379	0.2315
	{WMC x FANIN}	0.2944	0.2852	0.2939	0.2964	0.2903	0.2691	0.2923	0.2973
	{Qi x CBO}	0.2460	0.2537	0.2551	0.2500	0.2540	0.2503	0.2500	0.2523
	{WMC x CBO}	0.2762	0.2792	0.2816	2782	0.2833	0.2792	0.2702	0.2859

En termes de prédiction, les apprenants dopés (BOOSTING et BAGGING) donnent de meilleures performances pour l'algorithme C4.5. La régression logistique ne répond pas aux ensembles classificateurs BOOSTING et BAGGING. Selon le couple de métriques considéré, les performances lors de la validation avec 10x70/30 des modèles dopés chutent ou augmentent. En effet, les erreurs passent (par exemple) de 0.2369 à 0.2020 (soit un gain de 8.5 %) pour le modèle {Qi x FANIN} utilisant l'algorithme C4.5 dopé, tandis que pour le couple {WMC x CBO}, l'erreur passe de 0.3242 à 0.3383 pour le C4.5 boosté. Ce résultat peut s'expliquer par le surapprentissage observé de la technique de BOOSTING sur les données d'entrainement. Notons, par contre, que pour le C4.5 le BAGGING stabilise les résultats et améliore relativement toutes ses performances.

Table 51: Taux d'erreur des algorithmes d'apprentissage pour POI.

		Sans BOSTING / BAGGING				BOOSTING		BAGGING	
KNN	{PxG}	FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.1676	0.3158	0.3019	0.297	0.0642	0.3393	0.1546	0.3053
	{WMC x FANIN}	0.3017	0.3111	0.3103	0.3017	0.189	0.2929	0.1341	0.2892
	{Qi x CBO}	0.1629	0.3241	0.3224	0.3343	0.1117	0.3393	0.1443	0.317
	{WMC x CBO}	0.1667	0.318	0.3028	0.298	0.1555	0.2944	0.1611	0.3022
C4.5		FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.1574	0.2898	0.2944	0.284	0.0074	0.3164	0.1313	0.2817
	{WMC x FANIN}	0.216	0.2848	0.2785	0.2588	0.1574	0.3037	0.1611	0.2663
	{Qi x CBO}	0.149	0.3176	0.3168	0.3035	0.0084	0.3164	0.1155	0.2836
	{WMC x CBO}	0.1704	0.2873	0.286	0.2924	0.1099	0.3108	0.1415	0.2845
LOG		FITTING	10x70/30	10-FV	LOOV	FITTING	10x70/30	FITTING	10x70/30
	{Qi x FANIN}	0.2998	0.2997	0.3075	0.3017	0.3026	0.3102	0.3007	0.3003
	{WMC x FANIN}	0.2905	0.2901	0.2991	0.2914	0.2952	0.2944	0.2896	0.283
	{Qi x CBO}	0.3082	0.3074	0.3131	0.3101	0.3074	0.308	0.3073	0.3056
	{WMC x CBO}	0.3054	0.3139	3084	0.3073	0.3063	0.3003	0.3091	0.2957

Dans la table 51, nous remarquons que dans les colonnes FITTING les taux d'erreurs sont relativement plus faibles en général (plus petits que 30 %, ce qui était prévisible), soit une exactitude globale supérieure à 70 % en général. Ces colonnes donnent une idée du degré d'ajustement (fitting) de l'apprenant sur les données. Elles donnent aussi un aperçu du lien entre les couples de métriques considérées et les choix des développeurs quant aux classes à tester explicitement. Le BOOSTING améliore sensiblement ces taux d'erreur passant (par exemple) de 0.1574 à 0.0074 pour le couple {Qi x FANIN} avec le C4.5. Ce qui est aussi prévisible vu que le BOOSTING, dans son approche, tend à réduire le vote des mauvaises classifications. Le BAGGING est moins performant que le BOOSTING quand les tests s'effectuent sur les données d'entrainement. On peut trouver l'explication de cette contre-performance dans la construction des échantillons du BAGGING, où les observations sont pigées uniformément. Lors des validations croisées avec 10x70/30, l'algorithme C4.5 performe bien et s'améliore avec le BAGGING pour le couple de métriques {Qi x CBO}. Pour ce modèle, l'erreur passe de 0.3176 à 0.2836. Les erreurs issues de ces techniques de validation sont légèrement plus élevées. La validation du C4.5 avec le couple {WMC x FANIN} donne la meilleure performance avec une erreur de 0.2785 pour la validation croisée, et de 0.2588 avec le LOOV. Le BAGGING appliqué aux modèles C4.5 améliore significativement les performances de classification des apprenants pour certains couples de métriques. Pour d'autres couples, le BOOSTING et le BAGGING n'ont aucun effet sur le

taux d'erreur, et cela pour toutes les métriques : c'est le cas de la régression logistique (corrélation entre les jeux de données lors du dopage BOOSTING / BAGGING).

Du point de vue de l'orientation des tests, l'apprentissage automatique basé sur les métriques suggérées par le modèle de risque montre qu'il est possible de se servir de l'information sur les tests pour construire des modèles basés sur des couples d'attributs (l'un lié à la probabilité de faute et l'autre lié à la gravité de faute) proposant des ensembles tests à 70% identiques à ceux que proposerait un responsable de test connaissant le système.

Nous allons, dans ce qui suit, étudier les performances des ensembles de classes logicielles explicitement testées et celles des ensembles des classes logicielles proposés automatiquement par l'analyse du risque pour un système donné. L'objectif sera de voir lequel des ensembles couvre le mieux les classes du système contenant des fautes graves (sur le plan sévérité par rapport au logiciel). Nous allons, en d'autres termes, évaluer l'avantage réel d'utiliser les ensembles suggérés par les modèles de risque. En effet, jusqu'ici nous avons étudié les correspondances entre les ensembles tests suggérés automatiquement par les différentes techniques (basées sur le risque), et ceux proposés par les responsables de test.

9.6 Évaluation empirique des performances des tests

Dans cette section, nous évaluons les performances de l'orientation des tests basée sur le risque en estimant l'efficacité des ensembles tests suggérées automatiquement. Nous abordons l'efficacité des tests en termes de capacité des ensembles tests suggérés à cibler des classes pouvant contenir des fautes sévères pour le système. Nous comparons, ensuite, l'efficacité de ces ensembles aux ensembles tests explicites, fournis par les responsables des tests. Nous proposons une démarche empirique axée sur l'utilisation des informations sur les fautes et leurs sévérités, disponibles dans les dépôts logiciels pour estimer le risque que représente chaque classe logicielle.

9.6.1 Statistiques descriptives

Nous avons choisi des systèmes pour lesquels nous avons accès à l'information sur les fautes, les sévérités associées, les ensembles tests proposés par les responsables de test, ainsi que les codes sources des logiciels. C'est ainsi que nous avons étudié 5 versions majeures du logiciel ANT (présenté à la section 3.1) regroupant au total de 1694 classes logicielles (table 52). Afin de parvenir à associer les sévérités aux fautes et les fautes aux classes logicielles pour chaque version, nous avons analysé les répertoires des fautes, les forums de discussions (des développeurs) s'y rattachant, les patchs apportés pour les fixer ainsi que les différents niveaux de sévérité qui ont été attribués aux bogues. Le niveau de sévérité d'un bogue est établi par les responsables du projet/des tests à la découverte d'une faute, et avant de l'assigner au développeur pour correction. Le développeur peut éventuellement revoir la cote de sévérité établie s'il juge que la faute est plus ou moins sévère que prévu. Au total, 526 fautes ont été découvertes dans les 5 versions. Elles sont réparties en 6 niveaux de sévérité: triviale, mineure, normale, majeure, critique et bloquante. La répartition des fautes est décrite graphiquement par les camemberts de la figure 27.

Table 52: Statistiques descriptives des 5 versions d'ANT.

	#CL	%CT	#LOC	%LOCT	#Fautes
ANT 1.3	126	27.78%	37 935	35.62%	135
ANT 1.4	178	29.21%	54 196	43.31%	50
ANT 1.5	293	32.76%	87 047	45.82%	179
ANT 1.6	352	36.65%	113 260	51.34%	61
ANT 1.7	745	25.90%	208 653	43.78	101

Nous remarquons, à partir de la table 52, que le nombre de classes logicielles (#CL) et le nombre de lignes de code (#LOC) croissent régulièrement. Ils passent de 126 (contenant 37 935 lignes de code) à la version 1.3 à 745 (contenant 208 653 lignes de code) pour la version 1.7. Le nombre de fautes trouvées (#Fautes) ne suit pas nécessairement cette tendance avec un maximum de 179 fautes découvertes à la version 1.5 et un minimum de 50 à la version 1.4. Le pourcentage de classes explicitement testées (%CT) et le nombre de lignes de code qu'elles représentent (%LOCT) diffèrent d'une version à l'autre. La version 1.6 semble la plus explicitement couverte par les tests avec 36.65% de classes

logicielles contenant près de 51 000 lignes de code. Les taux de test explicite des classes logicielles varient autour de 30%.

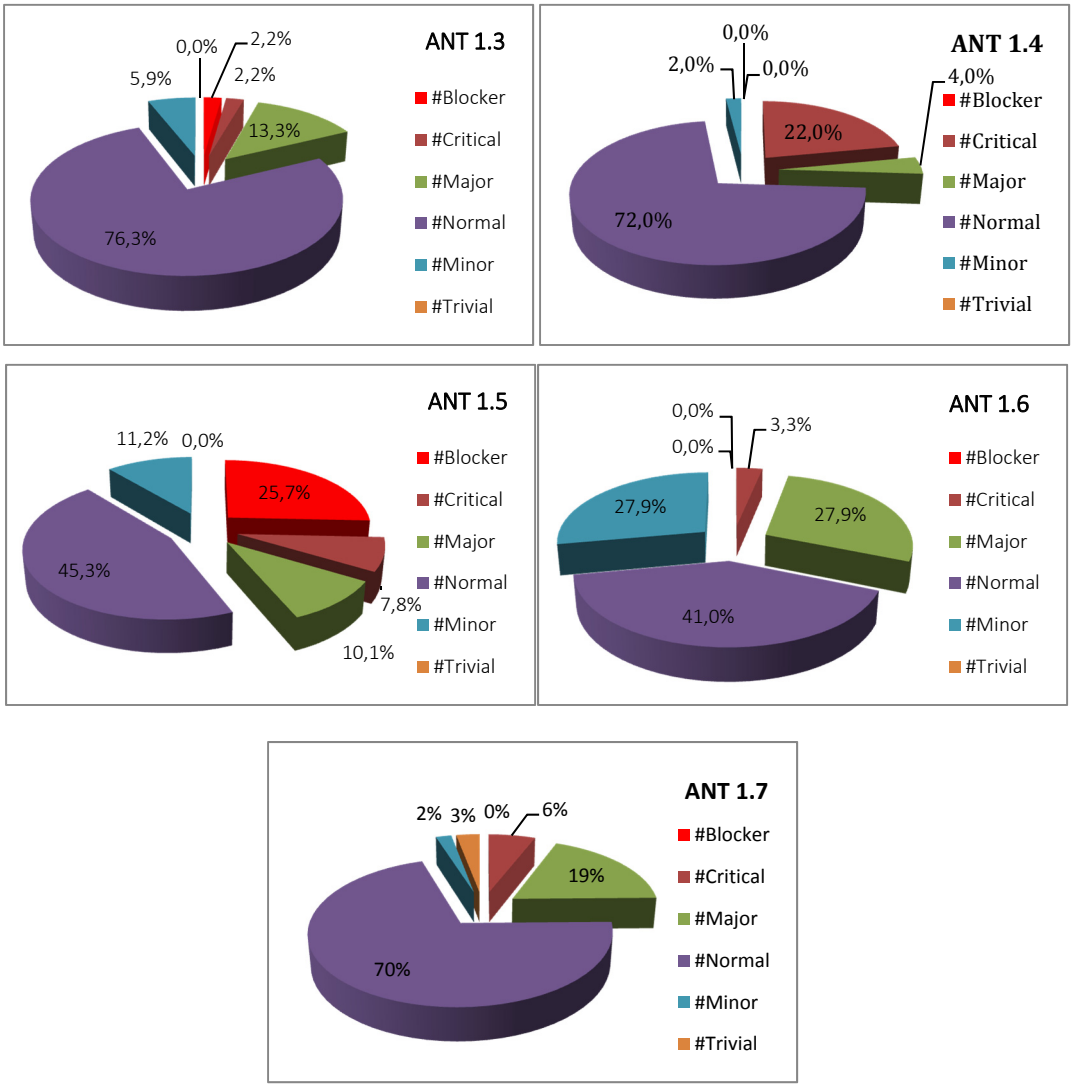


Figure 27: Répartitions des fautes trouvées en fonction des sévérités associées.

La répartition des 526 fautes découvertes dans les systèmes en fonction de leurs sévérités (figure 27) indique que la plupart des fautes ont été assignées au niveau de sévérité moyen par les responsables. Le niveau de sévérité moyen contient environ 70% des fautes trouvées dans les versions 1.3, 1.4 et 1.7 et autour de 40% de fautes trouvées pour les versions 1.5 et 1.6. Notons que la version 1.7 est la seule à contenir des fautes de niveau trivial (3), alors que les fautes bloquantes ont uniquement été découvertes dans les

versions 1.3 (2%) et 1.5 (25%). Les fautes critiques varient entre 2.2% pour 1.3 à 22% pour la 1.4.

Nous avons sélectionné des paires de métriques issues des familles P et G décrites précédemment (figure 21) parmi les attributs WMC, LOC, CBO, FANIN et les indicateurs de qualité Qi. La table 53 fournit les statistiques descriptives en termes de min., max., somme, moyenne (μ) et écart-type (σ) de ces métriques pour les classes de l'ensemble des 5 versions du système ANT. Elles regroupent près de 501 000 lignes de code.

Table 53: Statistiques descriptives des métriques utilisées.

	Qi	WMC	LOC	CBO	FANIN
Obs	1694	1694	1694	1694	1694
Min	0	0	0	0	0
Max	1	120	4541	499	427
Somme	1141.45	18 334	501 090	18 598	12 100
μ	0.674	10.823	295.803	10.979	7.143
σ	0.366	11.696	417.600	22.279	24.944

9.6.2 Estimation du risque empirique d'une classe logicielle

Dans les systèmes analysés, certaines classes contiennent des fautes qui ont été découvertes lors des tests ou de l'utilisation du logiciel. Ces fautes n'ont pas toutes les mêmes sévérités sur le système. En effet, pendant que certaines fautes sont mineures et nécessitent juste des corrections superficielles, d'autres peuvent être bloquantes et empêcher le système de fonctionner. Les (6) différents niveaux de sévérités, attribués par les responsables et réévalués par les développeurs (assignés aux fautes) dans le cas de ANT, sont récupérés lors de l'analyse des données issues du dépôt d'ANT. Pour tenir compte des dimensions de probabilité (fréquence) et sévérité des fautes associées à une classe logicielle, nous avons introduit la notion de risque empirique associé à une classe logicielle.

Le risque empirique pour une classe C est la somme du nombre de fautes découvertes dans cette classe pondérée par leurs niveaux de sévérité:

$$RE(C) = \sum_{i=1}^{6} S_i * N_i$$

N_i est le nombre de fautes de sévérité *i* découvertes dans la classe *C*,

S_i est le poids accordé au niveau de sévérité *i* (de triviale: 1 à bloquante: 6).

Plus une classe présente des fautes sévères, plus son risque empirique est élevé. Ce coefficient accorde une cote à une classe logicielle en tenant compte à la fois des fautes qui y ont été découvertes et de leurs sévérités. Il s'agit d'une caractéristique propre à une classe, issue de son historique de fautes et de sévérités associées (par les développeurs et les responsables).

Le RE d'une classe est un entier positif qui est relatif au système analysé du fait de la dimension de la sévérité qu'il contient (qui est relatif au logiciel analysé). Une valeur nulle pour une classe indique qu'aucune faute n'y a encore été découverte et donc n'a présenté (à ce jour) aucun risque. Une forte valeur (relativement aux autres classes du même système) peut suggérer 3 cas de figure: (1) Le cas extrême où la classe présente plusieurs fautes de grandes sévérités. Il peut s'agir de classe complexe et critique vis-à-vis du système dans le sens où elle est sollicitée par plusieurs autres composantes. (2) Le cas de forte fréquence de fautes, mais peu sévères, pouvant correspondre à une classe très complexe, mais peu sollicitée. (3) Le cas de rares découvertes de fautes, mais très sévères pouvant correspondre à une classe très sollicitée, mais peu complexe.

Nous évaluerons le total du risque empirique des ensembles tests proposés par les responsables des tests et des ensembles tests suggérés par les algorithmes (d'orientation automatique) basés sur le risque. L'objectif est de comparer et de déceler les ensembles qui maximisent la couverture du risque empirique total.

9.6.3 Estimation du risque théorique d'une classe logicielle

Afin d'orienter l'effort de test en nous basant sur l'analyse du risque, nous ordonnons les classes par rapport aux risques relatifs qu'elles présentent selon l'approche du risque présentée précédemment. En se basant sur les modèles de risque construits avec les couples {Qi, FANIN}, {Qi, CBO}, {WMC, FANIN}, {WMC, CBO}, {LOCC, FANIN} et {LOC, CBO}, nous évaluons les risques des classes grâce à la distance euclidienne, en utilisant leurs rangs. Les rangs sont sans dimensions et permettent de tenir compte des valeurs relatives des métriques des classes les unes par rapport aux autres sans être influencés par l'effet de

la taille des variables. Rappelons que dans la projection plane selon les rangs, présentée dans la section 9.4.1, le coin supérieur droit correspond au point de risque maximum (point M figure 28). Plus une classe est proche de ce point, plus elle devrait (selon notre approche) faire l'objet de tests approfondis et/ou de restructuration. Nous utiliserons la distance entre une observation (classes logicielles projetées selon les rangs) et ce point de repère comme estimation inverse du risque théorique que représente cette classe. Ce risque est uniquement basé sur les attributs de code source du logiciel.

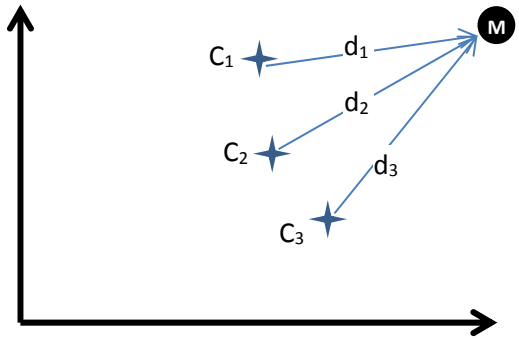


Figure 28: Distance entre les classes et le point de risque maximum (risque théorique).

Pour notre analyse, les classes C_i seront projetées dans le repère selon les rangs des couples (de métriques) considérés. Pour chacune des 5 versions du système ANT (de 1.3 à 1.7), nous avons compilé les métriques Qi, FANIN, CBO et WMC pour chacune des classes des différentes versions du système ANT. Nous avons ensuite déterminé les rangs des métriques, avant de projeter les classes dans le repère. L'idée d'utiliser les rangs reflète le fait que nous avons besoin juste d'une relation d'ordre qualitative pour estimer le risque relatif d'un composant par rapport à l'autre et pouvoir ainsi ordonner (orienter) l'effort global de test vers les composants aux rangs les plus élevés. Il ne s'agit pas de donner une évaluation absolue du risque associé à ce composant. Il serait, cependant, possible d'étendre ce modèle en pondérant les axes de probabilité et de gravité pour accorder plus d'importance à la dimension gravité ou à la dimension probabilité, suivant les préoccupations de l'équipe de développement et les métriques utilisées pour capturer ces dimensions.

Pour une classe logicielle donnée, et selon le modèle utilisant le couple de métrique {m_1, m_2}, nous définissons le risque théorique (relatif) par:

$$RT_{(m_1,m_2)}(C) = \sqrt{\left(rm_1(C) - \max_i(rm_1(C_i))\right)^2 + \left(rm_2(C) - \max_i(rm_2(C_i))\right)^2}$$

Où $rm_k(C)$ est le rang de la valeur de la métrique m_k pour la classe C par rapport aux autres classes du système. Le vecteur $\begin{pmatrix} \max_i(rm_1(C_i)) \\ \max_i(rm_2(C_i)) \end{pmatrix}$ forme les coordonnées du point M (point de risque maximum) dans le repère de projection. Ce point n'est pas nécessairement atteint par une observation. Plus *RT* est petit (la projection est proche du point *M)*, et plus grand est le risque que représente cette classe pour le système. En triant les classes logicielles de manière croissante selon la valeur de *RT*, on peut orienter l'effort de test qui minimise le risque théorique total pour un système. Pour évaluer l'intérêt de la métrique *RT*, nous allons à présent étudier le lien entre cette métrique de risque théorique et le risque empirique calculé plutôt (issu des dépôts de logiciel).

9.6.4 Analyse de corrélations

Nous avons effectué une analyse de corrélation entre le risque théorique et le risque empirique pour chaque modèle et pour chaque version du système ANT. L'objectif est de vérifier les hypothèses nulle (H0) et alternative (H1) suivantes:

- H0: Une classe logicielle de risque théorique élevé n'a pas un risque empirique plus élevé qu'une classe logicielle de risque théorique plus faible.
- H1: Une classe logicielle de risque théorique élevé a un risque empirique plus élevé qu'une classe logicielle de risque théorique plus faible.

Table 54 Corrélations de Spearman entre les risques empiriques et les risques théoriques.

	R_e(ANT1.3)	R_e(ANT1.4)	R_e(ANT1.5)	R_e(ANT1.6)	R_e(ANT1.7)
RT(LOC, FANIN)	**-0.550**	**-0.461**	**-0.470**	**-0.183**	**-0.249**
RT(QI, FANIN)	**-0.440**	**-0.421**	**-0.386**	**-0.150**	**-0.199**
RT(WMC, FANIN)	**-0.441**	**-0.419**	**-0.407**	**-0.156**	**-0.217**
RT(WMC, CBO)	**-0.530**	**-0.420**	**-0.446**	**-0.196**	**-0.291**
RT(Qi, CBO)	**-0.501**	**-0.408**	**-0.434**	**-0.205**	**-0.273**
RT(LOC,CBO)	**-0.626**	**-0.454**	**-0.523**	**-0.231**	**-0.328**

La table 54 résume les valeurs des corrélations entre les risques empiriques des différentes versions d'ANT et les risques théoriques issus des différents modèles de risque. Les valeurs en gras représentent les corrélations significatives (*valeur-p* ≥ α = 5%). Les corrélations sont toutes en gras (donc significatives au sens α). Les signes négatifs des corrélations montrent que plus une classe logicielle projetée est proche du point M du repère (risque théorique maximum), plus le risque empirique de cette classe est grand (la classe présente plus de fautes de plus grandes sévérités). Ce qui conforte l'hypothèse H1 que le modèle tel que présenté capture bien le risque.

Dans la table 54, le modèle construit avec le couple {LOC x CBO} présente les meilleures corrélations, ce qui peut s'expliquer par le fait que d'un côté, la métrique intrinsèque de taille (LOC) capture les informations de complexité qui ne sont pas uniquement liées à la complexité cyclomatique, mais aussi à la multiplication des données, des variables, des opérateurs qui peuvent influer sur la probabilité de faute. D'un autre côté, la métrique de couplage CBO capture aussi bien le flux entrant que sortant de par sa définition (en comparaison avec le FANIN qui ne capture que le flux entrant). CBO est fortement associée à la propagation de fautes et potentiellement à leurs sévérités [46,51].

9.6.5 Efficacité des tests orientés par le risque

Nous allons, à présent, évaluer le risque empirique couvert par les tests orientés par les différentes variantes (selon les couples de métriques) du modèle de risque, et celui ciblé par les tests effectués explicitement par les responsables des tests du logiciel ANT. L'objectif est de déterminer le gain qu'aurait obtenu un responsable de test en termes de risque empirique couvert, dans le cas où il aurait utilisé les ensembles tests fournis automatiquement par nos approches. Quel pourcentage de risque pourrait-il

potentiellement couvrir par les tests avec ces modèles ? Pour ce faire, nous avons calculé pour chaque version *v* du système ANT, le risque empirique total de l'ensemble test ($\mathrm{T}(exp)$ contenant *N* classes logicielles) proposé par les responsables des tests. Nous avons ensuite suggéré automatiquement les 6 ensembles tests $\mathrm{T}(m_1, m_2)$ formés des *N* premières classes les plus à risque, calculés selon le risque théorique pour le couple de métriques {m_1, m_2} et ordonné de manière croissante. Pour ces 6 ensembles tests de *N* classes, nous avons calculé le risque empirique total associé. Nous noterons $Re(\mathrm{T})$ la somme des risques empiriques associés à un ensemble test T.

$$Re(\mathrm{T}) = \sum_{C \in \mathrm{T}} RE(C)$$

Pour chaque version, le calcul de $Re(\mathrm{T}(m_1, m_2))$ permet de déterminer les risques empiriques couverts par les différents ensembles tests proposés automatiquement et de les comparer à celui $Re(\mathrm{T}(\mathrm{exp}))$ des classes explicitement testées. Finalement, nous avons calculé la fraction de risque empirique $FRe(\mathrm{T})$ que représente le risque empirique total de l'ensemble test T par rapport au risque empirique total des classes de la version *v* d'ANT considérée.

$$FRe(\mathrm{T}) = \frac{Re(\mathrm{T})}{\sum_{C \in \mathrm{V}} RE(C)}$$

V étant l'ensemble des classes d'une version *v* de ANT

La table 55 résume les résultats obtenus pour chaque version d'ANT.

Table 55: Risques ciblés par les stratégies de test.

		T(exp)	T(LOC, FANIN)	T(Qi, FANIN)	T(WMC, FANIN)	T(WMC,CBO)	T(Qi, CBO)	T(LOC, CBO)
ANT 1.3	Card(T)	35	35	35	35	35	35	35
	Re(T)	212	402	366	358	382	346	394
	FRe(T)	29.61 %	56.15 %	51.12 %	50.00 %	53.35 %	48.32 %	55.03 %
ANT 1.4	Card(T)	52	52	52	52	52	52	52
	Re(T)	126	254	242	250	250	230	266
	FRe(T)	37.28 %	75.15 %	71.60 %	73.96 %	73.96 %	68.05 %	78.70 %
ANT 1.5	Card(T)	96	96	96	96	96	96	96
	Re(T)	1012	1384	1254	1396	1410	1324	1518
	FRe(T)	45.92 %	62.79 %	56.90 %	63.34 %	63.97 %	60.07 %	68.87 %
ANT 1.6	Card(T)	129	129	129	129	129	129	129
	Re(T)	98	152	158	146	178	204	198
	FRe(T)	32.45 %	50.33 %	52.32 %	48.34 %	58.94 %	67.55 %	65.56 %
ANT 1.7	Card(T)	193	129	129	129	129	129	129
	Re(T)	314	336	287	320	371	355	394
	FRe(T)	58.26 %	62.34 %	53.25 %	59.37 %	68.83 %	65.86 %	73.10 %

Les résultats montrent que les ensembles tests T(exp) proposés par les responsables des tests (dans presque toutes les versions) ont couvert beaucoup moins de risque empirique que les ensembles suggérés par les différents modèles d'analyse du risque . Une exception est observée pour ANT 1.7 et le modèle fourni par le couple {Qi, FANIN}. En effet:

- pour ANT 1.3: La couverture du risque empirique est passée de 29.6 % pour l'ensemble test suggéré par les responsables des tests à 56,15 avec le modèle {LOC x FANIN}.
- pour ANT 1.4: La couverture du risque empirique est passée de 37.28 % pour l'ensemble test suggéré par les responsables des tests à 78.70 % avec le modèle {LOC x CBO}.
- pour ANT 1.5: La couverture du risque empirique est passée de 45.92 % pour l'ensemble test suggéré par les responsables des tests à 88.87 % avec le modèle {LOC x CBO}.
- pour ANT 1.6: La couverture du risque empirique est passée de 32.45 % pour l'ensemble test suggéré par les responsables des tests à 65.56 % avec le modèle {Qi x CBO}.
- pour ANT 1.7: La couverture du risque empirique est passée de 58.26 % pour l'ensemble test suggéré par les responsables des tests à 73.10 % avec le modèle {Qi x CBO}.

Une couverture maximale de risque empirique de 78.7 % est observée pour le modèle de risque basé sur LOC et CBO avec le système ANT1.4. Pour ANT1.3, nous observons une faible couverture de risque de l'ensemble de test proposé par les responsables des tests (moins de 30 %), alors que l'ensemble suggéré par le modèle {LOC x

FANIN} couvre plus de 56 % du risque empirique total des classes de cette version. C'est la plus grande amélioration de couverture de risque empirique observée dans cette analyse.

Nous remarquons aussi une amélioration sur la couverture de risque des ensembles tests *T(exp)* (proposés par les responsables de test) au cours du temps. Cela peut s'expliquer, entre autres, par l'expérience acquise par l'équipe au fil des versions. La version 1.6 fait exception et constitue, après vérifications de l'historique des changements, une évolution majeure d'ANT. Les résultats suggèrent que l'utilisation de l'approche d'analyse du risque permet d'améliorer grandement le ciblage des classes à haut risque (de 20 à 40 %), ce qui peut être très utile lorsque les ressources affectées au test sont limitées ou que la complexité du système ne permet pas des tests exhaustifs.

9.6.6 Risque pour la validité

Menées sur plus de 500 000 lignes de code, nos expérimentations sur le modèle de risque donnent des résultats certes significatifs, mais leur généralisation peut être limitée par des facteurs groupés en risque de validité interne, externe et conceptuelle.

L'un des facteurs internes pouvant partiellement influencer les résultats se trouve dans le processus de test. En effet, le fait de savoir qu'une classe logicielle va être testée peut inciter le développeur à prêter plus d'attention dans l'écriture du code. Il en résulte des classes contenant moins de fautes et biaisant notre manière d'évaluer le risque empirique des ensembles explicitement testés par les responsables de tests.

Par ailleurs, des risques externes peuvent limiter directement la généralisation des résultats issus de cette démarche empirique. En effet, cette expérimentation a été menée sur différentes versions d'un même système. Or, nous savons que d'un système à l'autre, les critères de sévérité et la manière de les assigner aux fautes peuvent changer ce qui pourrait avoir un impact sur les résultats obtenus.

Le processus de classification non formelle des fautes en différents niveaux de sévérité introduit un autre risque sous forme de menace à la validité conceptuelle, pouvant biaiser notre manière de mesurer la sévérité. En effet, d'une part l'assignation des valeurs 1 à 6 est arbitraire et pourrait être changée pour d'autres valeurs reflétant mieux les échelles

d'impacts des fautes. D'autre part, l'absence de formalisme dans le processus d'affectation des niveaux de sévérité aux fautes dans les projets logiciels peut biaiser le processus d'assignation. Finalement, l'indicateur de risque empirique est dynamique. Il peut évoluer dans le temps avec la découverte de nouvelles fautes. La dernière version analysée (la 1.7) est sortie en décembre 2006, même si nous pensons avoir laissé assez de temps pour la découverte de la plupart des fautes, il n'empêche que certaines n'ont été découvertes que récemment (2014).

9.7 Conclusion sur le risque et l'orientation de l'effort de test

Nous avons analysé, dans cette section, 9 systèmes logiciels dont 5 versions du système ANT. Nous avons mis en place une approche d'évaluation du risque des classes logicielles basée uniquement sur les métriques de code source. L'étude nous a permis de grouper les métriques en 2 familles selon qu'elles influencent les facteurs de probabilité ou de gravité de faute. Nous avons abordé la gravité sous l'angle de l'impact de la faute sur le système. Nous avons ensuite extrait des métriques de chaque famille pour construire des modèles prédicteurs automatiques grâce à 3 algorithmes d'apprentissage. Nous avons dopé les algorithmes, et les avons validés par 4 différentes techniques de validation croisée. Les résultats suggèrent qu'avec des couples de métriques, il est possible de construire des modèles capables de proposer des ensembles de test correspondant à environ 70% aux ensembles tests que proposent les responsables des tests. Nous avons, ensuite, proposé une technique simple d'orientation en nous basant sur la proximité du point de risque maximal. Nous avons évalué cette approche en utilisant des données empiriques de 5 versions d'ANT. Les résultats suggèrent que cette approche automatique d'orientation basée sur le risque le plus élevé cible mieux les fautes pondérées par leur sévérité. L'étude menée dans cette section est exploratoire et les résultats auxquels elle aboutit doivent être considérés aussi comme exploratoires. De plus amples investigations sont nécessaires afin de tirer des conclusions plus générales.

CONCLUSION GÉNÉRALE

Rappel de la Problématique

Dans cette thèse, nous avons investigué la problématique de l'orientation de l'effort des tests unitaires dans les systèmes orientés objet. L'objectif était de proposer un modèle permettant d'orienter l'effort de test vers les composants les plus à risque à l'aide des métriques de code source logiciel. Nous avions divisé la problématique en plusieurs sous-problématiques ayant chacune un objectif spécifique précis. Le premier objectif était d'évaluer la testabilité sous l'angle de l'effort d'écriture et de construction des cas de test unitaires. Le deuxième objectif était d'étudier les métriques des tests unitaires utilisées pour caractériser l'effort de test. Le troisième objectif visait la prédiction des différents niveaux d'effort de test à l'aide de métriques de code source. Une fois l'effort évalué, les liens établis et les métriques de code source déterminées, nous nous sommes fixés pour dernier objectif, de construire des modèles d'orientation de l'effort pour les tests unitaires.

Contributions

Les différents sous-objectifs nous ont conduit à explorer différentes facettes des liens entre les métriques logicielles orientées objet (incluant les Qi) et l'effort de test unitaire. Ces études, qui ont été menées sur une dizaine de systèmes logiciels orientés objet *open source* totalisant plus d'un demi-million de lignes de code, ont suggéré l'existence de liens significatifs ainsi que la possibilité de prédiction de l'effort de test unitaire par les métriques de code source et les indicateurs de qualité. L'étude des métriques des tests unitaires à travers leur redondance, leur volatilité et leur pertinence a montré l'existence de deux dimensions capturées par deux métriques de test peu volatiles que sont la taille des suites de test en termes de lignes de code et le nombre d'invocations de méthodes. Nous avons raffiné l'analyse de l'effort de test en essayant de prédire les différents niveaux d'effort, des niveaux obtenus par deux techniques distinctes de regroupement automatique des classes tests. Les résultats ont montré que les indicateurs de qualité, la taille et la complexité permettaient de prédire correctement ces différents niveaux.

La découverte de liens entre les métriques logicielles et l'effort de test et ses différents niveaux nous a permis de proposer quelques modèles d'orientation de l'effort de test à partir des métriques logicielles. La première approche d'orientation s'appuie sur les algorithmes de classification automatique. Elle a mis en évidence la possibilité de réutilisation des règles apprises automatiquement d'un système dans le cadre de l'orientation de l'effort de test à d'autres systèmes. La deuxième approche est basée sur l'analyse du risque au niveau des classes logicielles. Le risque a été évalué à l'aide des métriques logicielles en estimant la probabilité de survenance de fautes dans une classe et leurs gravités en termes d'impact sur le système. Les études empiriques qui ont suivi ont été menées sur près de 1.2 million de lignes de code à travers 10 systèmes logiciels orientés objet et ont montré que: (1) Cette technique permettait de proposer automatiquement des ensembles tests intéressants lors des choix des classes qui seront explicitement testées en ce sens qu'ils sont proches des ensembles de tests explicitement proposés par les testeurs. (2) Les ensembles de tests issus de l'orientation par l'approche du risque performent mieux que les ensembles de tests explicites en termes de couverture globale des fautes pondérées par leurs sévérités (risque empirique).

Certaines contributions ont fait l'objet de publications dans des actes de conférences internationales et des revues spécialisées [56, 57, 58, 122, 123, 143, 144] tandis que d'autres sont en cours d'évaluation ou de rédaction.

Limitations

Ces résultats sont certes assez significatifs au regard du nombre et de la taille des logiciels analysés, mais doivent néanmoins être considérés comme exploratoires. Des facteurs cachés peuvent, en effet, concourir à expliquer certains résultats (et limites observées). Dans ce cadre, nous pouvons remarquer que les systèmes open source utilisés présentent certaines spécificités dans leurs processus de développement et dans leurs équipes qui peuvent influencer les résultats obtenus lors de nos investigations. Par exemple, le choix du nombre et des classes à tester explicitement est dans la plupart du temps, laissé au bon vouloir des développeurs. Cela peut entrainer des classes

partiellement testées et des systèmes ayant peu de classes testées, ce qui en retour, peut influencer les résultats obtenus dans nos différentes expérimentations. Par ailleurs, certaines classes sont testées de manière indirecte à travers l'invocation de leurs méthodes par d'autres classes. Cette dimension est complètement ignorée dans notre processus d'identification des classes tests associées aux classes logicielles. Cette étape étant dans notre cas, basée sur la nomenclature des classes.

La généralisation de nos résultats nécessite des investigations complémentaires notamment l'analyse de systèmes propriétaires *(non-open source).* Les processus de développement et de suivi de ces systèmes (souvent) industriels se font en général selon des normes de qualité plus rigoureuses comparées aux normes de qualité des systèmes *open source*. Les équipes de développement peuvent aussi être plus restreintes. Par ailleurs, les domaines d'applications des logiciels analysés peuvent aussi impacter les résultats et restreindre leurs portées. La diversification des sources de données d'analyse avec notamment le choix de logiciels propriétaires et l'intégration (du facteur) du domaine d'application des logiciels analysés dans nos investigation faciliterait la généralisation des résultats observés. Dans nos analyses, nous nous sommes limités au langage orienté objet JAVA. Même si ce dernier est aujourd'hui une référence en matière de langage de développement, mature, populaire et intègre tous les artéfacts de la technologie orientée objet, il reste néanmoins que nos résultats peuvent être biaisés par la considération de systèmes uniquement développés dans ce langage, ce qui contribue à restreindre leur portée.

Finalement, notons que des biais existent quant à la construction des expérimentations. Ils concernent principalement l'estimation du risque et l'aspect dynamique de l'effort de test. La gravité estimée des fautes dans le modèle de risque que nous avons proposé se limite à l'impact de celles-ci sur le système. Nous ignorons de ce fait tous les facteurs externes importants qui pourraient amplifier la gravité d'une faute sur un système logiciel en exploitation, notamment la criticité du domaine d'application du logiciel. Par ailleurs, la testabilité telle que considérée tout au long de nos investigations concerne uniquement l'effort de construction et d'écriture des tests unitaires. De ce fait,

tous les aspects dynamiques qui requièrent des sources d'informations autres que les métriques de code source ont été ignorés. La prise en compte de ces facteurs aiderait à avoir des meilleures estimations des facteurs que l'on souhaite étudier.

Recherches futures

Ces résultats entrouvrent, cependant, les possibilités d'utiliser les métriques logicielles et l'expérience des développeurs dans l'orientation de l'effort global des tests unitaires. En effet, se basant sur ces résultats, il serait intéressant de développer des outils basés sur le Cloud et les techniques d'analyse du Big data actuelles (impliquant les algorithmes d'intelligence artificielle) pour bâtir une nouvelle génération d'outils d'orientation de tests intégrés aux environnements de développement et basés sur la collaboration communautaire des développeurs logiciels. Cette piste sera notre futur axe de recherche.

Les limitations soulevées précédemment orienteront aussi nos futures recherches vers l'analyse de l'effort des tests unitaires sous d'autres angles, notamment en ce qui a trait à son aspect dynamique. Pour cela, l'utilisation de métriques dynamiques et la considération du facteur temporel dans nos modèles et nos mesures permettront d'orienter nos investigations sur les aspects dynamiques des tests unitaires, aspects qui viendront complémenter les résultats obtenus dans le cadre des analyses statiques effectuées dans cette thèse.

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] Hoang P., Software Reliability, Wiley Encyclopedia of Electrical and Electronics Engineering, DOI: 10.1002/047134608X.W6952, 1999.

[2] Prowell S.J., Trammell C.J., Linger R.C., et Poore J.H., Cleanroom Software Engineering: Technology and Process, published by Pearson Education, 1999.

[3] Cukier K., Mayer-Schoenberger V., Big Data: La révolution des données est en marche, publié par ROBERT LAFFONT/BOUQUINS/SEGHER ISBN:9782221144510, 2014.

[4] The Manifesto for Agile Software Development, http://agilemanifesto.org, Visité le 15 février 2016.

[5] Agile Alliance, https://www.agilealliance.org, Visité le 15 février 2016.

[6] Rick G., Object Oriented Technologies: Opportunities and Challenges, published by Idea Group Inc. (IGI), ISBN 193070870X, 9781930708709, 1999.

[7] Perlis A.J., Sayward F. et Shaw M., Software Metrics: An Analysis and Evaluation, Volume 5 MIT Press, ISBN 0262160838, 9 780 262 160 834, 1981.

[8] Pressman R.S., Software engineering: A pratictioner's approach, 5th edition. McGraw-Hill higher Education, 2001.

[9] NIST (Natonal Institute of Stadards & Technology), The Economic Impact of Inadequate Infrastructure for Software Testing planning, Report 2002-2003.

[10] Myers G.J., Sandler C. et Badgett T., The Art of Software Testing, published by Wiley, ITPro collection, ISBN:9 781 118 133 156, 2011.

[11] Tahat, L. Vaysburg B., Korel B., et Bader A., Requirement-based automated black-box test generation, 25th Annual International Computer Software and Applications Conference, Chicago, Illinois pp. 489–495, 2001.

[12] Astels D., Test-driven Development: A Practical Guide, Prentice Hall PTR, The Coad series, ISBN:9 780 131 016 491, 2003.

[13] Ciortea L., Zamfir C., Bucur S. Chipounov V. et Candea G., Cloud9: A software testing service, ACM SIGOPS Operating Systems Review Archive, Volume 43, Issue 4, January 2010.

[14] Lazić L., Software Testing Optimization by Advanced Quantitative Defect Management, Computer Science and Information Systems, Vol. 7, No. 3, 459-487, 2010.

[15] Rothermel G., Untch R.H., Chu C. et Harrold M.J., Prioritizing Test Cases for Regression Testing, IEEE Transactions on Software Engineering, vol. 27, No. 10, October 2001.

[16] Borland Together, http://www.borland.com/en-GB/Home, Visité le 15 février 2016.

[17] Laurent S.T. et Andrew M., Understanding Open Source and Free Software Licensing. O'Reilly Media. p. 4. ISBN 9780596553951, 2008.

[18] Karolak D.W., Software Engineering Risk Management: Finding Your Path through the Jungle, IEEE Computer Society Press, June 1995.

[19] Willam R. C., Pandelio G.J et Behrens S.G., Sofware Risk Evaluation (SRE) Method Description V2. Technical Report, Carnegie Mallon Software Engineering Institute, Pittsburgh, PA, December 1999.

[20] Foo S. et Muruganatham A., Software Assessment Model ICMIT, IEEE, pp-536-544, 2000.

[21] IEEE16085: Systems and Software Engineering Life Cycle Processes Risk Management, The Institute of Electrical and Electronics. Engineers, Inc., 2006.

[22] ISO/IEC 27001:2005 Specification for Information Security Management. International Organization for Standardization (ISO) /International, Electrotechnical Commission (IEC), 2005.

[23] Kontio J. et Basili V.R., Empirical Evaluation of a Risk Management Method. SEI conference of Risk Management. Atlantic City NJ, US 1997.

[24] Russell S. et Norvig P., Artificial Intelligence: A modern approach 2nd edition, Prentice Hall Series in Artificial Intelligence, 2003.

[25] Fenton N. et Neil M., Software Metrics and Risk. FESMA 99, 2nd European Software Measurement Conference, 1999.

[26] Gupta D. et Mohd S., Software risk assessment Estimation Model, International Conference on Computer Science and Information Technology, 2008.

[27] Sherif J. S., Metrics for software risk management, WESCON/96 IEEE, ISMN#O-7803-3274-1, pp.507-513, 1996.

[28] Menezes J.J., Gusmao C. et Moura H., Indicators and metrics for risk assessment in software projects: A mapping study, Proc. 5th Experimental Software Engineering Latin American Workshop (ESELAW 2008), 2008.

[29] IEEE Standard Computer Dictionary, A Compilation of IEEE Standard Computer Glossaries, IEEE Std. 610-1991, doi:10.1109/IEEESTD.1991.106963, 1991.

[30] Boehm B.W., Software Engineering Economics, Prentice Hall, 1981.

[31] Lipow M., Prediction of Software Failures, Journal of Systems and Software, vol. 1, no. 1, pp. 71-75, 1979.

[32] Khoshgoftaar T.M., Improving usefulness of software quality classification models based on Boolean discriminant functions, Thirteenth international symposium on software reliability engineering, IEEE Computer Society, Annapolis, MD, USA, 2002.

[33] Menzies T. et Di Stefano J.S., How good is your blind spot sampling policy?, 8th IEEE international symposium on high-assurance systems engineering, IEEE Computer Society, Tampa, FL, USA, pp. 129–138, 2004.

[34] Mende T., et Koschke R., Revisiting the evaluation of defect prediction models, In Proceedings of the 5th international conference on predictor models in software engineering (Vancouver, BC, Canada), PROMISE '09 (pp. 1–10), New York, NY: ACM. 2009.

[35] Pandey A.K. et Goyal N.K., Fault Prediction Model by Fuzzy Profile Development of Reliability Relevant Software Metrics, International Journal of Computer Applications (0975 – 8887) Volume 11– No.6, December 2010.

[36] Basili V.R., Briand L.C. et Melo W.L., A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering. vol. 22, no. 10, pp. 751-761, October. 1996.

[37] Chidamber S.R. et Kemerer C.F., A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[38] Briand L., Wust J., Daly J. et Porter D., Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems, J. Systems and Software, vol. 51, no. 3, pp. 245-273, 2000.

[39] El Emam K., Melo W. et Machado J., The Prediction of Faulty Classes Using Object-Oriented Design Metrics, J. Systems and Software, vol. 56, no. 1, pp. 63-75, 2001.

[40] kaur K., Parvinder K.S, Amandeep B.S., Early software fault prediction using real time defect data, Second International Conference on Machine Vision, pp 243-245, 2009.

[41] NASA IV &V Facility, Metric Data Program, available from http://MDP.ivv.nasa.gov, Visité le 15 février 2016.

[42] Aggarwal K.K., Singh Y., Kaur A., et Malhotra R., Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A Replicated Case Study, Software Process Improvement and Practice, vol. 14, no. 1, pp. 39-62, 2009.

[43] Rathore S.S. et Gupta A., Investigating Object-Oriented Design Metrics to Predict Fault-Proneness of Software Modules, Software engineering (CONSEG), CSI 6th International Conference, 2012.

[44] www.promisedata.org, Visité le 15 février 2016.

[45] Gyimothy T., Ferenc R. et Siket I., Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction, IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897-910, October 2005.

[46] Shatnawi R., A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems, IEEE Transactions On Software Engineering, Vol. 36, No. 2, March/April 2010.

[47] Bender R., Quantitative Risk Assessment in Epidemiological Studies Investigating Threshold Effects, Biometrical J., vol. 41, no. 3, pp. 305-319, 1999.

[48] Catal C., Sevim U. et Diri B., Clustering and Metrics Thresholds Based Software Fault Prediction of Unlabeled Program Modules, 6th International Conference on Information Technology: New Generations, 2009.

[49] Shatnawi R., Improving Software Fault-Prediction for Imbalanced Data, International Conference on Innovations in Information Technology (IIT), 2012.

[50] Subramanyan R. et Krisnan M.S., Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Trans. Software Eng., vol. 29, no. 4, pp 297-310, April 2003.

[51] Zhou Y. et Leung H., Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults, IEEE Transaction Software Engineering, vol. 32, no. 10, pp. 771-789, October. 2006

[52] Sandhu P.S., Singh S. et Budhija N., Prediction of Level of Severity of Faults in Software Systems using Density Based Clustering, International Conference on Software and Computer Applications IPCSIT vol. 9, IACSIT Press, Singapore, 2011.

[53] Iliev M., Karasneh B., Chaudron M.R.V. et Essenius E., Automated Prediction of Defect Severity Based on Codifying Design Knowledge Using Ontologies, Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), First International Workshop, P.7-11, IEEE, 2012.

[54] Ostrand T.J., Weyuker E.J. et Bell R.M., Predicting the Location and Number of Faults in Large Software Systems, IEEE Trans. Software Eng., vol. 31, no. 4, pp. 340-355, April 2005.

[55] Ostrand T.J. et Weyuker E.J., The Distribution of Faults in a Large Industrial Software System, Proc. Int'l Symposium. Software Testing and Analysis, pp. 55-64, 2002.

[56] Toure F., Badri M. et Lamontagne L., A metrics suite for JUnit test code: a multiple case study on open source software, Journal of Software Engineering Research and Development, Springer, December 2014.

[57] Badri M. et Toure F., Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes, Journal of Software Engineering and Applications (JSEA), 2012.

[58] Toure F., Badri M. et Lamontagne L., Towards a metrics suite for JUnit Test Cases. In Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE Vancouver, Canada. Knowledge Systems Institute Graduate School, USA pp 115–120, 2014.

[59] XUNIT, JUNIT: http://junit.org, CUNIT: http://cunit.sourceforge.net, NUNIT: http://www.nunit.org, QUNIT: https://qunitjs.com, Visité le 15 février 2016.

[60] CodePro, https://developers.google.com/java-dev-tools/codepro/doc, Visité le 15 février 2016.

[61] Rothermel G., Untch R.H., Chu C. et Harrold M.J., Test case prioritization: an empirical study International Conference on Software Maintenance, Oxford, UK, pp. 179–188, September 1999.

[62] Jatain A., et Sharma G., A Systematic Review of Techniques for Test Case Prioritization, International Journal of Computer Applications (0975 – 8887) Volume 68– No.2, April 2013.

[63] Malishevsky A.G., Ruthruff J.R., Rothermel G. et Elbaum S., Cost-cognizant Test Case Prioritization, Technical Report TRUNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska – Lincoln, 2006.

[64] Yua Y. T. et Laub M. F., Fault-based test suite prioritization for specification-based testing, Information and Software Technology Volume 54, Issue 2, Pages 179–202, February 2012.

[65] Carlson R. et Denton A., A clustering approach to improving test case prioritization: An industrial case study, 27th IEEE International Conference on Software Maintenance (ICSM), 2011.

[66] Elbaum S., Rothermel G., Kanduri S. et Malishevsky A.G., Selecting a cost-effective test case prioritization technique, Software Quality Control, 12(3):185–210, 2004.

[67] Mirarab S. et Tahvildari L., A prioritization approach for software test cases on Bayesian networks, In FASE, LNCS 4422-0276, pages 276–290, 2007.

[68] Rothermel G., Harrold M.J., Ronne J. et Hong C., Empirical Studies of Test-Suite Reduction, In Journal of Software Testing, Verification, and Reliability, Vol. 12, No.4, 2002.

[69] Leon D. et Podgurski P., A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases, Proc. Int'l Symp. Software Reliability Eng., pp. 442-453, 2003.

[70] Bryce R. et Memon A.M., Test Suite Prioritization by Interaction Coverage, Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DOSTA 2007). ACM: New York, 2007.

[71] Walcott K.R., Soffa M.L., Kapfhammer G.M. et Roos R.S., Time aware test suite prioritization. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006). ACM Press, New York, 1–12, 2006.

[72] Wong, W., Horgan, J., London, S., et Agrawal, H., A study of effective regression in practice, Proceedings of the 8th International Symposium on Software Reliability Engineering, November, p. 230–238, 1997.

[73] Mirarab S. et Tahvildari L., An Empirical Study on Bayesian Network-based Approach for Test Case Prioritization, International Conference on Software Testing, Verification, and Validation, 2008.

[74] Kim J. et Porter A., A history-based test prioritization technique for regression testing in resource constrained environments, In Proceedings of International Conference on Software Engineering, May 2002.

[75] Qu B., Nie C., Xu B. et Zhang X., Test case prioritization for black box testing, 31st Annual International Computer Software and Application conference, COMPSAC, 2007.

[76] Lin C.T., Chen C.D., Tsai C.S. et Kapfhammer G. M., History-based Test Case Prioritization with Software Version Awareness, International Conference on Engineering of Complex Computer Systems, 2013.

[77] Liu W.N., Huang C.Y., Lin C.T. et Wang P.S., An evaluation of applying testing coverage information to historical-value-based approach for test case prioritization, Proc. Asia-Pacific Symp. Internetware, pp. 73-81, December 2001.

[78] http://sir.unl.edu, Visité le 15 février 2016.

[79] Bach J., The challenge of good enough software, Presented at American Programmer, 1995.

[80] Bach J., Heuristic Risk-Based Testing, Software Testing and Quality Engineering Magazine, 11/1999.

[81] Amlan S., Risk Based Testing and Metrics: Risk Analysis Fundamentals and Metrics for software testing including a Financial Application case study, 5th International Conference EuroSTAR '99, Barcelona, Spain, November 8 - 12, 1999.

[82] Esouza E., Gusmao C., Alves K., Venâncio J. et Melo R., Measurement and Control for Risk-based Test Cases and Activities, 10th IEEE Latin American Test Workshop – LATW, 2009.

[83] Jorgesen L.K.U., A Software Tool for Risk-based Testing, Graduation Work, Norwegiant University of Science and Technology-Norway, 2004.

[84] Chen Y. et Probert R.L., A Risk-based regression test selection strategy, Fast Abstract, 14th ISSRE, ACM, 2003.

[85] Rosenberg L.H., Stapko R., et Gallo A., A Risk Based Object Oriented Testing, 24th Annual Software Engineering Testing Workshop, NASA SEW24, 1999.

[86] Ray M. et Mohapatra D.P., Prioritizing Program elements: A pretesting effort to improve software quality, International Scholarly Research Network, ISRN Software Engineering volume, 2012.

[87] Commons-IO, https://commons.apache.org/proper/commons-io/, Visité le 15 février 2016.

[88] Commons-Math, https://commons.apache.org/proper/commons-math/, Visité le 15 février 2016.

[89] Joda-Time, http://www.joda.org/joda-time/, Visité le 15 février 2016.

[90] Dbunit, http://dbunit.sourceforge.net/, Visité le 15 février 2016.

[91] Log4J, https://logging.apache.org/log4j/1.2/index.html, Visité le 15 février 2016.

[92] JFreeChart, http://www.jfree.org/jfreechart/, Visité le 15 février 2016.

[93] Apache Ivy, http://ant.apache.org/ivy/, Visité le 15 février 2016.

[94] Apache Lucene, https://lucene.apache.org/core/, Visité le 15 février 2016.

[95] Apache Ant, http://ant.apache.org/, Visité le 15 février 2016.

[96] Apach POI, https://poi.apache.org/, Visité le 15 février 2016.

[97] Henry S. et Kafura K., Software structure metrics based on information flow, IEEE Transactions on Software Engineering, 7(5): 510–518, 1981.

[98] Rook, P., Software Reliability Handbook, publication Kluwer Academic Publishers, Elsevier's Applied Science, p.462, 1990.

[99] Briand L.C., Wüst J. et Lounis H., Using Coupling Measurement for Impact Analysis in Object-Oriented Systems, in the Proceedings of IEEE, International Conference Software Maintenance (ICSM), pp. 475-482, Oxford, England, August/September 1999.

[100] McCabe T. J., A Complexity Measure, IEEE Transactions on Software Engineering: 308–320, 1976.

[101] Bruntink M., et Deursen A.V., Predicting Class Testability using Object-Oriented Metrics, 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE, 2004.

[102] Bruntink M. et Van Deursen A., An Empirical Study into Class Testability, Journal of Systems and Software, Vol. 79, No. 9, pp. 1219-1232. doi:10.1016/jss. 2006.02.036, 2006.

[103] Binder R.V., Design for testability in object-oriented systems, Communications of the ACM, vol. 37, no. 9, 1994.

[104] Singh Y., Kaur A., et Malhotra R., 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, pp 1012–1017, 2008.

[105] Halstead M.H., Elements of Software Science, Elsevier/North-Holland, New York, NY, USA, 1977.

[106] Eclipse, www.eclipse.org, Visité le 15 février 2016.

[107] Mockus A., Nagappan N., et Dinh-Trong T.T., Test coverage and post-verification defects: A multiple case study, In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09), pp 291–301, 2009.

[108] Rompaey B.V. et Demeyer S., Establishing traceability links between unit test cases and units under test, Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)., pp 209–218, 2009.

[109] Qusef A., Bavota G., Oliveto R., De Lucia A. et Binkley D., SCOTCH: test-to-code traceability using slicing and conceptual coupling, Proceedings of the International Conference on Software Maintenance (ICSM'11), 2011.

[110] Apache BugZilla, https://bz.apache.org/bugzilla/, Visité le 15 février 2016.

[111] XLSTAT, https://www.xlstat.com/fr/, Visité le 15 février 2016.

[112] TANAGRA, http://eric.univ-lyon2.fr/~ricco/tanagra/en/tanagra.html, Visité le 15 février 2016.

[113] D. Hosmer et S. Lemeshow, Applied Logistic Regression, Wiley-Interscience, 2nd edition, 2000.

[114] Cattell R.B., The scree test for the number of factors. Multivariate Behavioral Research Volume 1, Issue 2, 1966.

[115] MacQueen J. B., Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press. pp. 281–297. MR 0214227, Zbl 0214.46201, 1967.

[116] Fisher W.D., On grouping for maximum homogeneity, J. Amer. Statist. Assoc., Vol.53, pp780-798, 1958.

[117] Quinlan J. R., C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.

[118] Freund Y. and Schapire R.E., A decision-theoretic generalization of on-line learning and and application to boosting, Journal of computer and system sciences, 55(1):119–139, 1997.

[119] Badri M., Badri L., et Toure F., Empirical analysis of object oriented design metrics: towards a new metric using control flow paths and probabilities, Journal of Object Technology, vol.8, no. 6, p. 123–142, 2009.

[120] Henderson-Sellers B., Object-Oriented Metrics Measures of Complexity, Prentice-Hall, Upper Saddle River, 1996.

[121] Aggarwal K.K., Singh Y., Kaur A., et Malhotra R, Empirical study of object-oriented metrics, Journal of Object Technology, 5(8), 149–173, 2006.

[122] Badri M. et Touré F., Empirical analysis for investigating the effect of control flow dependencies on testability of classes, in Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE '11), 2011.

[123] Badri M. et Touré F., Evaluating the Effect of Control Flow on the Unit Testing Effort of Classes: An Empirical Analysis, Advances in Software Engineering Volume 2012 (2012), Article ID 964064, 2012.

[124] Badri M., Drouin, N. et Touré F., On Understanding Software Quality Evolution from a Defect Perspective: A Case Study on an Open Source Software System, Proceedings of the IEEE International Conference on Computer Systems and Industrial Informatics, Sharjah, UAE, December 18-20, 2012.

[125] Drouin N., Badri M. et Touré F., Analyzing Software Quality Evolution using Metrics: An Empirical Study on Open Source Software, Journal of Software, Vol. 8, No 10, 2462-2473, doi:10.4304/jsw.8.10.2462-2473, October 2013.

[126] Baudry B., Le Traon B. et Sunyé G., Testability Analysis of a UML Class Diagram, 9th International Software Metrics Symposium, Sydney, September 3-5, 2003.

[127] Baudry B., Le Traon B., Sunyé G., et Jézéquel J.M., Measuring and Improving Design Patterns Testability, 9th International Software Metrics Symposium, Sydney, September 3-5, 2003.

[128] Yeh P. L. et Lin J. C., Software testability measurement derived from data flow analysis, 2nd Euromicro Conference on Software Maintenance and Reengineering, Florence, March 8-11, 1998.

[129] IEEE, IEEE Standard Glossary of Software Engineering Terminology, IEEE CSP, New York, 1990.

[130] Fenton N. et Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.

[131] Freedman R.S., Testability of software components, IEEE Transactions on Software Engineering, vol. 17, no. 6, pp. 553–564, 1991.

[132] Voas J.M., PIE: A dynamic failure-based technique, IEEE Transactions on Software Engineering, vol. 18, no. 8, p. 717–727, 1992.

[133] Voas J.M. et Miller K.W., Semantic metrics for software testability, Journal of Systems and Software, Vol. 20, 1993.

[134] ISO/IEC 9126, Software Engineering Product Quality, 1991.

[135] Gao J. et Shih M.C., A component testability model for verification and measurement, in Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC '05), pp. 211–218, July 2005.

[136] Badri L., Badri M. et Touré F., Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems, JSEA Eds., Advances in Software Engineering, Communications in Computer and Information Science, Vol. 117, Springer, Berlin, 2010.

[137] Badri L., Badri M. et Touré F., An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes, International Journal of Software Engineering and Its Applications, Vol. 5, No. 2, pp. 69-85, 2011.

[138] Gupta V., Aggarwal K.K. et Singh Y., A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability, Journal of Computer Science, Vol. 1, No. 2, pp. 276-282. doi:10.3844/jcssp.2005.276.282, 2005.

[139] Briand L.C., Labiche Y. et Sun H., Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code, Software - Practice and Experience, vol. 33 (7), pp. 637-672, 2003.

[140] Mueller J.H. et Schuessler K.F., Statistical Reasoning in Sociology, Boston, Houghton Mifflin Company, 1961.

[141] Wnuk K., Regnell B., et Berenbach B., Scaling up requirements engineering - exploring the challenges of increasing size and complexity in market-driven software development, Proceedings of the 17th International Working Conference on Requirements Engineering, Foundation for software quality, Springer-Verlag, Berlin, Heidelberg, REFSQ'11, pp 54–59, 2011.

[142] Toure F., Indicateur de qualité pour les systèmes orientés objet : vers un modèle unifiant plusieurs métriques, Comme exigence partielle de la maîtrise en mathématiques et informatique appliquées, UQTR, Décembre 2007.

[143] Badri M., Toure F. et Lamontagne L., Predicting Unit Testing Effort Levels of Classes: An Exploratory Study based on Multinomial Logistic Regression Modeling, Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15), March 2015.

[144] Toure F., Badri M. et Lamontagne L., Predicting different levels of the unit testing effort of classes using metrics: a multiple case study, International Journal of Software Engineering and Knowledge Engineering (IJSEKE), soumis aprés demande de modifications mineures, Septembre 2015.

[145] Bernoulli D., Exposition of a New Theory on the Measurement of Risk, Econometrica, Vol. 22, No. 1, pp. 23-36. jan., 1954.

[146] Voas J.M. et Miller, K.W., Software Testability: The New Verification, IEEE Software, 12(3), 1995.

[147] Le Traon Y. et Robach C., Testability analysis of co-designed systems', Proceedings of the 4th Asian Test Symposium, ATS, IEEE Computer Society, Washington DC, 1995.

[148] Joshi K., Belwal R.C. et Mishra S., An Algorithmic Approach to Predict Fault Propagation and Defects in Dependent Modules based on Coupling, International Journal of Computer Applications, Volume 68– No.12, April 2013.

[149] Yeresime S., Lov K., et Santanu K. R., Statistical and Machine Learning Methods for Software Fault Prediction Using CK Metric Suite: A Comparative Analysis, ISRN Software Engineering, Article ID 251083, 15 pages, 2014. doi:10.1155/2014/251 083, 2014.

[150] Catal C. et Diri B., Software Fault Prediction with Object-Oriented Metrics Based Artificial Immune Recognition System, Product-Focused Software Process Improvement, pp. 300–314, 2007.