Organisation des développeurs open-source et fiabilité logicielle

Matthieu Foucault

▶ **To cite this version:**

Matthieu Foucault. Organisation des développeurs open-source et fiabilité logicielle. Génie logiciel [cs.SE]. Université de Bordeaux, 2015. Français. <NNT : 2015BORD0219>. <tel-01255901v2>

HAL Id: tel-01255901

https://tel.archives-ouvertes.fr/tel-01255901v2

Submitted on 1 Feb 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Académie de Bordeaux
Université De **B**ordeaux
Sciences et **T**echnologies

Thèse

présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Organisation des développeurs *open-source* et fiabilité logicielle

par

Matthieu Foucault

Soutenue le 30 Novembre 2015, devant le jury composé de :

Président du jury
Guy Melançon, professeur .... Université de Bordeaux, France

Directeur de thèse
Xavier Blanc, professeur .... Université de Bordeaux, France

Co-Directeur de thèse
Jean-Rémy Falleri, maitre de conférences .... Bordeaux INP, France

Rapporteurs
Marianne Huchard, professeur .... Université de Montpellier, France
Stéphane Ducasse, directeur de recherche .... INRIA Lille Nord Europe, France

Examinateurs
Benoît Baudry, chargé de recherche HDR .... INRIA Rennes, France

Remerciements

Avant d'attaquer le vif du sujet, il me semble indispensable de consacrer quelques lignes aux personnes sans qui ce document ne pourrait pas être entre vos mains (ou sur votre écran). Les trois années qui ont conduit à l'écriture de ce manuscrit (qui techniquement est un tapuscrit) ont été riches en enseignements, autant personnels que professionnels, et nombre de personnes ont participé à la richesse de cette expérience.

Je tiens à commencer par remercier tout particulièrement les deux personnes qui m'ont guidé tout au long de cette thèse, Xavier Blanc et Jean-Rémy Falleri. Si ces trois années de thèse se sont aussi bien passées, c'est en grande partie grâce à eux, grâce à la liberté qu'ils m'ont accordé, autant scientifique que géographique, et à leur participation active aux différents travaux de recherche que nous avons mené. La réussite d'une thèse dépend en grande partie de la qualité de l'encadrement; j'ai eu la chance d'avoir un duo de directeurs de thèse exceptionnels et complémentaires.

Je remercie Benoît Baudry, Stéphane Ducasse, Marianne Huchard et Guy Melançon d'avoir pris le temps de s'intéresser à mes travaux et d'avoir accepté de faire partie de mon jury de thèse. Je suis honoré d'avoir eu des échanges avec des chercheurs aussi brillants lors de la journée qui a conclu cette aventure.

J'adresse mes plus profonds remerciements à mes parents, sans qui toutes ces années d'études n'auraient pas été possible. Je vous serai toujours reconnaissant pour tout le soutien que vous m'avez apporté pendant toutes ces années.

Je remercie Alan, sans qui les journées passées au bureau seraient beaucoup trop calmes. Je te souhaite bonne chance pour la fin de ta thèse, qui m'a l'air très bien partie. Je remercie également Xavier, avec qui les journées passées au bureau étaient beaucoup trop intenses. C'était génial! Merci à tous les deux pour toutes ces pauses café et ces parties de ping-pong qui ont aidé à se défouler.

Je veux adresser un grand merci à Marc Palyart. J'espère que nos collaborations scienti-

fiques ne font que commencer et j'espère qu'on aura l'occasion de partager à nouveau un bureau (bien que je ne suis pas sûr que ce soit bon pour notre productivité).

Je voudrais remercier tous ceux que je peux compter parmi mes amis: Xavier, Alan et Marc, susmentionnés, mais aussi Milan et Veronica, Sébastien, Vivien et Nico. Je remercie Sébastien pour tout le soutien apporté lors des quatre dernières années et pour sa participation à la relecture de ce document. J'ai des remerciements tout particulier à adresser aux personnes avec qui j'ai vécu pendant quatre ans, mes colocataires Martin et Célia, qui ont eu la bonne idée de s'échapper avant la période de fin de rédaction. Merci pour ces super moments passés en collocation, et à Martin pour m'avoir montré que c'était trop cool de faire une thèse.

Je voudrais remercier tous mes collègues du LaBRI et en particulier les membres du thème Génie Logiciel. Je souhaite bonne chance à mes collègues doctorants Elyas, Hanyang et William pour la fin de leur thèse, et je tiens à remercier Laurent et Floréal pour les enseignements qu'ils m'ont apporté. Merci à Cédric et Arthur, à qui je souhaite bonne chance dans l'aventure qu'est la création d'entreprise. Je tiens à remercier tout le personnel administratif du LaBRI pour l'aide qu'ils m'ont apporté afin de réaliser les différentes démarches.

J'ai également eu le plaisir de découvrir l'enseignement de l'informatique pendant cette thèse, et je veux remercier à nouveau Xavier, Jean-Rémy, Laurent et David pour m'avoir permis de faire toutes ces heures d'enseignement. Je tiens à remercier les étudiants de l'ENSEIRB-MATMECA et de l'université de Bordeaux que j'ai pu encadrer pendant ces trois années, en espérant qu'ils ont autant apprécié ces cours et travaux dirigés que moi.

Enfin, je tiens à remercier les personnes avec qui j'ai eu l'occasion de collaborer à l'Université de Colombie Britannique, à Vancouver, en particulier Marc Palyart, encore une fois, et Gail Murphy, grâce à qui la suite de ma carrière de chercheur est bien lancée.

Pour conclure, à toutes les personnes que j'ai oubliées de mentionner dans ces remerciements, je m'excuse.

Table des matières

Remerciements i

Table des matières iii

1 Introduction 1

1.1 Fiabilité et métriques logicielles 2

1.2 Validation des métriques avec les projets *open-source* 3

1.2.1 Évaluation empirique des métriques 3

1.2.2 Fouilles de données dans les dépôts logiciels 3

1.2.3 Les projets *open-source* comme source de données 4

1.3 Problématiques de recherche et contributions 5

1.3.1 Méthodologie 6

1.3.2 Répartition des contributions et fiabilité logicielle 7

1.3.3 Turnover et fiabilité logicielle 8

1.4 Structure 10

2 État de l'art 11

2.1 Mesure de la fiabilité logicielle 12

2.1.1 Extraction des bogues et des *commits* correctifs 12

2.1.2 Informations additionnelles sur les bogues 13

2.2 Répartition des contributions 14

2.2.1 Mesurer les contributions des développeurs 14

2.2.2 Nombre de développeurs et fiabilité logicielle 17

2.2.3 Réseaux de contributions 18

2.2.4 Propriété du code source 19

2.3 *Turnover* . . . 21
2.3.1 Dans la sociologie des organisations . . . 22
2.3.2 Dans les plateformes de collaboration en ligne . . . 22
2.3.3 Dans le génie logiciel . . . 23
2.4 Synthèse de l'état de l'art . . . 24

3 Méthodologie 25
3.1 Aperçu de la méthodologie . . . 26
3.2 Critères de sélection des projets . . . 29
3.2.1 Gestionnaire de versions . . . 29
3.2.2 Architecture des branches de développement et de maintenance . . . 30
3.2.3 Autres critères de sélection . . . 31
3.3 Extraction des métriques . . . 32
3.3.1 Découpage du code source en modules . . . 32
3.3.2 Identification des alias des développeurs . . . 33
3.3.3 Extraction des *commits* correctifs . . . 34
3.3.4 Calcul des métriques de fiabilité . . . 36
3.3.5 Extraction de l'activité des développeurs . . . 36
3.3.6 Calcul des métriques d'organisation . . . 38
3.4 Analyse statistique des métriques . . . 38
3.4.1 Corrélation entre métriques . . . 39
3.4.2 Régression multiple et importance relative des métriques . . . 41
3.4.3 Synthèse . . . 42
3.5 Réplication des résultats . . . 43
3.5.1 Données . . . 43
3.5.2 Suite logicielle . . . 43
3.5.3 Projets sélectionnés . . . 44
3.6 Validité de la méthodologie . . . 45
3.6.1 *Pull requests* et revue de code . . . 45
3.6.2 Origine du code réusiné . . . 45
3.6.3 Découpage du code en modules . . . 45
3.6.4 *Commits* correctifs et bogues . . . 46

4 Répartition des contributions 47
4.1 Introduction . . . 48
4.2 Définitions des métriques . . . 49
4.2.1 Activité des développeurs . . . 49
4.2.2 Propriété du code . . . 50
4.2.3 Concentration de l'activité . . . 51
4.3 Théories . . . 52
4.4 Méthodologie . . . 54

4.5 Analyse des résultats . . . 54
4.5.1 Relation entre les métriques de répartition des contributions et de fiabilité . . . 56
4.5.2 Importance relative des métriques . . . 58
4.6 Validité de l'étude . . . 59
4.6.1 Contributeurs majeurs et mineurs . . . 59
4.6.2 Lignes de conduite de la répartition des contributions . . . 61
4.7 Limites et travaux futurs . . . 61

5 Turnover 63
5.1 Introduction . . . 64
5.2 Impact théorique de la rotation des effectifs . . . 65
5.2.1 Arrivées de développeurs . . . 65
5.2.2 Départs de développeurs . . . 65
5.2.3 Réorganisation interne du projet . . . 66
5.3 Définition des métriques . . . 66
5.3.1 Pré-requis . . . 66
5.3.2 Acteurs du *turnover* . . . 67
5.3.3 Définitions des métriques . . . 69
5.4 Problématiques de recherche . . . 70
5.5 Sélection des périodes de temps . . . 70
5.6 Résultats . . . 72
5.6.1 Rotation des effectifs à l'échelle du projet (**PR1**) . . . 73
5.6.2 Répartition des acteurs du *turnover* entre les modules (**PR2**) . . . 74
5.6.3 Relation avec la fiabilité logicielle (**PR3**) . . . 78
5.6.4 Importance relative des métriques . . . 81
5.7 Validité de l'étude . . . 83
5.8 Limites et travaux futurs . . . 83

6 Conclusion 85
6.1 Résumé des contributions . . . 85
6.2 Perspectives . . . 88
6.2.1 Motivations des développeurs . . . 88
6.2.2 Mesure précise de l'activité . . . 88

A Métriques de *turnover* : sélection des périodes 91

Table des figures 107

Liste des tableaux 109

Listings 111

CHAPITRE

1 Introduction

Ce chapitre introduit au lecteur le contexte, les motivations et les objectifs de notre thèse. Cette thèse s'appuie sur la nécessité pour les développeurs de produire des logiciels fiables et sur le coût de la maintenance permettant cette fiabilité. L'intérêt des métriques logicielles est alors présenté : ces métriques permettent, via des modèles de prédiction de bogues, d'orienter les développeurs sur les composants logiciels les plus susceptibles de contenir des bogues ; c'est sur ces composants logiciels que les opérations de maintenance telles que le développement de tests doivent être réalisées en priorité.

Nous avons choisi d'étudier des métriques relatives au procédé de développement et en particulier à l'organisation des développeurs. Nos contributions sont liées à la validation de ces métriques, qui est faite empiriquement, c'est-à-dire en observant la relation de ces métriques avec la fiabilité sur un ensemble de projets logiciels.

Sommaire

1.1 Fiabilité et métriques logicielles **2**

1.2 Validation des métriques avec les projets *open-source* **3**

1.3 Problématiques de recherche et contributions **5**

1.4 Structure **10**

1.1 Fiabilité et métriques logicielles

Le succès d'un logiciel passe par sa qualité, qui est perçue par les utilisateurs via différentes caractéristiques telles que sa performance et son bon fonctionnement. Un logiciel de mauvaise qualité ne satisfera pas les parties intéressées, utilisateurs ou développeurs, et nuira au succès du projet associé à ce logiciel [Ralph et Kelly, 2014]. L'aspect premier de la qualité aux yeux d'un utilisateur est la fiabilité, c'est-à-dire la capacité du logiciel à produire les fonctionnalités attendues [ISO, 2011; Kitchenham et Pfleeger, 1996]. Un logiciel ou un composant d'un logiciel ne produit pas les fonctionnalités attendues lorsqu'un dysfonctionnement a lieu. Un dysfonctionnement est causé lorsqu'une partie du code source contenant un bogue est exécutée [Kan, 2002].

Un des objectifs principaux d'un développeur étant de produire un programme ayant la meilleure fiabilité possible, il faut que le code source du programme contienne le moins de bogues possible. Notre objectif est donc de permettre aux développeurs de réduire le nombre de bogues présents dans leur logiciel de façon efficace. Cette aide aux développeurs peut prendre de nombreuses formes et peut avoir lieu avant ou après l'observation de dysfonctionnements.

Le pan de travaux qui nous intéresse a pour but d'empêcher l'apparition des dysfonctionnements logiciels en définissant des lignes de conduites ou des techniques permettant de réduire le nombre de bogues dans le code source. Parmi les outils et techniques développés pour aider à prévenir les fautes logicielles, on peut trouver des techniques relatives au procédé de développement, telles que la revue de code, l'accompagnement des nouveaux arrivants dans les projets logiciels [Canfora *et al.*, 2012] ou les techniques telles que l'*extreme programming* [Beck, 1999] ou le *test-driven development* [Beck, 2003]. L'efficacité de ces techniques est cependant fortement liée au contexte de chaque projet [Dyba et Dingsøyr, 2008; Fagerholm *et al.*, 2014].

En complément de ces techniques se trouvent les métriques logicielles, qui sont des indicateurs relatifs à des propriétés du code source du logiciel ou du procédé de développement. Ces métriques ont pour but de savoir quels comportements ont un impact sur la fiabilité logicielle afin de mettre en place de nouvelles lignes de conduite ou de prédire la localisation de bogues, permettant ainsi d'optimiser les opérations de maintenance du code source. On trouve classiquement deux types de métriques : les métriques de code et les métriques de procédé. Rahman et Devanbu suggèrent que les métriques de procédé sont meilleures que les métriques de code lorsqu'il s'agit de prédire les bogues des différents modules [Rahman et Devanbu, 2013]. Une raison évoquée est le fait que les métriques de procédé sont moins stagnantes que les métriques de code, ces dernières ayant tendance à rester figées au fil de l'évolution d'un logiciel. Par exemple, une classe ayant une grande complexité sera classifiée comme potentiellement défectueuse par une métrique de code, même si le code de cette classe n'a pas été modifié depuis de nombreuses versions du logiciel. La recherche relative à l'ensemble de métriques orientées sur le procédé de développement est relativement récente et de nombreux aspects de ce procédé n'ont

pas encore été étudiés [Radjenović *et al.*, 2013]. Nous avons donc choisi de nous intéresser aux métriques de procédé pour notre thèse et à leur relation avec la fiabilité logicielle. Ces métriques de procédé considèrent l'évolution du logiciel et les actions réalisées pour amener à la version du code source actuelle. Elles se basent par exemple sur l'hypothèse que les parties du logiciel ayant subi des modifications récentes sont les plus propices à la présence de bogues [Girba *et al.*, 2004] ou que le nombre ou l'expérience des développeurs ayant contribué à une portion de code source a un impact sur sa fiabilité [Weyuker *et al.*, 2008; Rahman et Devanbu, 2011]. La question qui se pose alors est la validation de ces métriques logicielles.

1.2 Validation des métriques avec les projets *open-source*

Lorsqu'une nouvelle métrique est développée il est nécessaire de la valider. Les contributions de notre thèse sont centrées sur la validation de métriques. Nous présentons donc ici le procédé permettant la validation des métriques et discutons de l'obtention des données nécessaires à cette validation.

1.2.1 Évaluation empirique des métriques

La recherche empirique en génie logiciel se base sur des observations répétées de phénomènes dans le but de valider une théorie [Kitchenham *et al.*, 2004]. Dans le cas des métriques logicielles, il s'agit d'observer des logiciels existants et de mesurer la relation entre les métriques et la fiabilité par le biais de tests de corrélation et de construction de modèles de prédiction de bogues. La répétition de ces observations est indispensable, en recherchant à chaque nouvelle étude des moyens d'améliorer la précision des mesures, la qualité des données extraites et la portée des conclusions en ajoutant de nouveaux logiciels à ces observations [Kitchenham *et al.*, 2004]. Un exemple probant d'une telle répétition est l'ensemble d'études consacrées aux métriques de design du code orienté objet définies par Chidamber et Kemerer [Chidamber et Kemerer, 1994], dont la relation avec la fiabilité a été étudiée à de nombreuses reprises depuis 1996 [Brito e Abreu et Melo, 1996], avec des métriques telles que NOC (le nombre de fils d'une classe) et DIT (la profondeur de l'arbre d'héritage) qui ont été le sujet d'au moins cinquante études sur une période de quinze ans [Radjenović *et al.*, 2013].

1.2.2 Fouilles de données dans les dépôts logiciels

Cette évaluation empirique des métriques est ancrée dans un champ de recherche appelé fouille de données dans les dépôts logiciels (ou *mining software repositories* en anglais). Les dépôts logiciels en question peuvent contenir l'historique du projet, le code source du logiciel et des informations sur les bogues identifiés dans le code source. Les

données extraites depuis ces dépôts logiciels permettent de réaliser les observations relatives à la relation entre métriques et fiabilité, mais aussi de mieux comprendre le procédé de développement et les interactions entre les développeurs de ces projets.

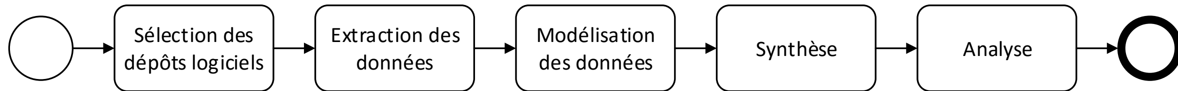

FIGURE 1.1 : Étapes du procédé de fouilles de données dans les dépôts logiciels [Hemmati *et al.*, 2013].

Le procédé de fouilles de données dans les dépôts logiciels se découpe généralement en cinq étapes, illustrées figure 1.1 [Hemmati *et al.*, 2013]. Dans chacune de ces étapes, un certain nombre de choix d'algorithmes ou de techniques d'extraction doit être réalisé afin de permettre la production de résultats de qualité. Ces étapes sont la sélection des dépôts logiciels appropriés, l'extraction des données et leur modélisation dans le but de représenter les concepts recherchés, la synthèse de ces données via les métriques logicielles et l'analyse statistique de ces données.

Un des objectifs des études du domaine *mining software repositories* est de développer de nouvelles heuristiques permettant la production de résultats de qualité. Ces heuristiques sont nécessaires par exemple lors de la modélisation des données lorsqu'on souhaite s'appuyer sur l'identité des individus ayant participé au projet : les actions de ces contributeurs sont enregistrées sous forme d'un alias (une adresse e-mail par exemple) et cet alias peut évoluer au cours du temps. Une heuristique est donc nécessaire pour lier les alias aux développeurs [Goeminne et Mens, 2013]. Le développement de ces heuristiques et de la méthodologie des études de fouilles de données dans les dépôts logiciels fait partie du processus de recherche empirique évoqué plus haut.

1.2.3 Les projets *open-source* comme source de données

Une des problématiques pour la validation de métriques logicielles est l'obtention de données relatives au développement de projets [Hassan, 2008]. Dans de nombreux cas, les entreprises ne souhaitent pas mettre à disposition les dépôts logiciels contenant ces données. Une autre possibilité est l'utilisation de projets académiques, développés par exemple par des étudiants, mais les historiques de ces projets ne sont pas aussi riches que ceux de logiciels ayant plusieurs versions et de nombreux utilisateurs. Une solution est l'observation de projets *open-source*, dont la particularité est que leur code source et dans la plupart des cas leur historique de développement est accessible librement en ligne.

Cette disponibilité des données se fait cependant au détriment de la facilité de leur extraction : contrairement au contexte industriel où certains outils internes à l'entreprise facilitent le suivi du développement du projet, l'extraction des données dans les projets

open-source requiert souvent une ou plusieurs étapes de pré-traitement afin de faire le lien entre les différents dépôts du projet. Ce paramètre est une explication possible de la faible proportion d'études concernant des projets *open-source*.

Le modèle de développement *open-source* a la particularité d'attirer la participation de nombreux contributeurs, qu'ils soient bénévoles ou employés par une entreprise sponsorisant le projet [Crowston *et al.*, 2008; Capiluppi *et al.*, 2012; Riehle *et al.*, 2014]. Contrairement au développement de logiciels fermés et propriétaires, les contributeurs de projets *open-source* ne sont pas nécessairement liés par une obligation contractuelle. De ce fait, le niveau de contribution des développeurs peut varier grandement d'un individu à l'autre, allant de l'ajout d'un unique *patch* à une participation continue au projet et à la direction de celui-ci.

Cette particularité est intéressante dans notre optique d'évaluer les métriques de procédé de développement. Cette organisation des développeurs spécifique aux projet *open-source* implique une distribution des contributions potentiellement différente de ce qui a pû être observé par le passé dans des projets propriétaires. De ce fait, les phénomènes impactant la fiabilité logicielle et les métriques utilisées pour les synthétiser ne sont pas nécessairement généralisables entre les projets propriétaires et *open-source*. De plus, cette absence d'obligation contractuelle pour un grand nombre de développeurs implique que l'ensemble des contributeurs d'un projet *open-source* a tendance à évoluer plus rapidement que dans le contexte des logiciels propriétaires, ce qui est un autre aspect pouvant impacter la fiabilité d'un logiciel.

Objectif de la thèse

L'objectif général de cette thèse est de contribuer à la validation de métriques de procédé en étudiant leur relation avec la fiabilité. Ces métriques, une fois validées, pourront être utilisées dans des modèles de prédiction de bogues ayant pour but de mieux orienter les efforts de maintenance des développeurs ou pourront permettre de mettre en place des lignes de conduite relatives au procédé de développement. Devant l'étendue de ce domaine, nous avons centré nos contributions sur un aspect du procédé de développement qui est l'organisation des développeurs et avons observé cette organisation dans des projets *open-source*.

1.3 Problématiques de recherche et contributions

Nous présentons ici les problématiques de recherche abordées dans cette thèse. Ces problématiques portent sur deux aspects de l'organisation des développeurs dans les projets *open-source* :

- lorsque nous avons commencé cette thèse, plusieurs études introduisant des métriques relatives à la répartition des contributions des développeurs ont été publiées

avec des résultats prometteurs concernant leur relation avec la fiabilité [Bird *et al.*, 2011; Posnett *et al.*, 2013]. Notre première problématique est donc de confirmer ces résultats en réalisant une nouvelle validation de métriques existantes ;

- les métriques de répartition des contributions ont une vue relativement statique de l'équipe de développement, étant donné qu'elles se basent sur l'activité de tous les développeurs participant au projet pendant une période donnée, sans prendre en compte l'évolution de l'équipe de développeurs. Notre seconde problématique est d'évaluer la relation entre cette évolution de l'équipe de développeurs, aussi appelée *turnover*, et la fiabilité logicielle, en définissant de nouvelles métriques de procédé.

Avant de pouvoir aborder ces problématiques liées à l'organisation des développeurs, nous devons également mettre en place une méthodologie permettant la validation des métriques logicielles.

1.3.1 Méthodologie

La validation des métriques liées à nos problématiques de recherche nécessite l'extraction d'informations depuis les gestionnaires de versions de projets *open-source* et l'étude de la relation entre les mesures d'organisation des développeurs et de fiabilité extraites. Ce processus de validation n'est pas inédit, mais un des fondements de la recherche empirique est d'améliorer à chaque itération l'efficacité de la méthodologie et la qualité des résultats produits par celle-ci [Kitchenham *et al.*, 2004].

Le domaine de la fouille de données dans les dépôts logiciels est relativement récent et de nombreux travaux de recherche ont pour but de permettre d'améliorer la qualité des données extraites depuis ces dépôts et la compréhension de ces données [Bird *et al.*, 2009a; Goeminne et Mens, 2013; Herzig *et al.*, 2013]. Une évolution a également lieu au niveau des outils utilisés par les développeurs des logiciels observés, comme par exemple le gestionnaire de versions Git [1] qui permet désormais une identification plus précise des contributeurs [Mens *et al.*, 2005; Bird *et al.*, 2009b].

Nous avons donc développé une méthodologie prenant en compte ces nouveaux éléments afin de construire un jeu de données répondant à un certain nombre d'exigences relatives à la qualité des informations extraites. Cette méthodologie définit les différentes étapes du procédé de fouilles de données dans les projets logiciels, incluant les critères de sélection des projets, les différents algorithmes de pré-traitement des données appliqués lors de leur extraction et les outils statistiques utilisés afin d'analyser les métriques extraites. Enfin, nous présentons notre jeu de données et le programme que nous avons développé permettant la reproduction et l'extension de nos résultats.

1. http://git-scm.com

1.3.2 Répartition des contributions et fiabilité logicielle

Lorsque le nombre de développeurs collaborant sur un même projet augmente, la coordination entre ceux-ci devient plus complexe, ce qui peut entrainer des délais et une diminution de la fiabilité du logiciel produit [Brooks, 1975]. Étant donné ces problématiques de coordination entre développeurs, une première hypothèse est que l'augmentation du nombre de développeurs a un impact négatif sur la fiabilité logicielle. Cependant, l'effet contraire n'est pas non plus exclu si l'on prend en compte les observations réalisées par Eric Raymond et la théorie qui en découle, qui est que l'augmentation du nombre de développeurs augmente la probabilité qu'un des développeurs détecte et corrige un bogue [Raymond, 1999].

La mesure de la répartition des contributions nécessite de prendre en compte non seulement le nombre de développeurs contribuant à chaque module, mais aussi la quantité de contributions apportée par chaque développeur. De telles mesures ont été définies précédemment [Bird *et al.*, 2011; Posnett *et al.*, 2013] en se basant sur l'hypothèse que le fait d'avoir un développeur majeur ayant un ratio élevé de contributions comparé aux autres développeurs permet de réduire le nombre de bogues, étant donnée la capacité de ce développeur à synchroniser et évaluer les contributions des développeurs moins expérimentés.

Cette hypothèse est illustrée par la figure 1.2, où l'on peut voir deux modules logiciels :

- le module *A* a un faible nombre de développeurs, et un développeur principal réalisant une large majorité des contributions au module ;
- le module *B* a un nombre plus élevé de développeurs et la plupart d'entre eux sont des développeurs « mineurs », réalisant moins de 5% des contributions au module.

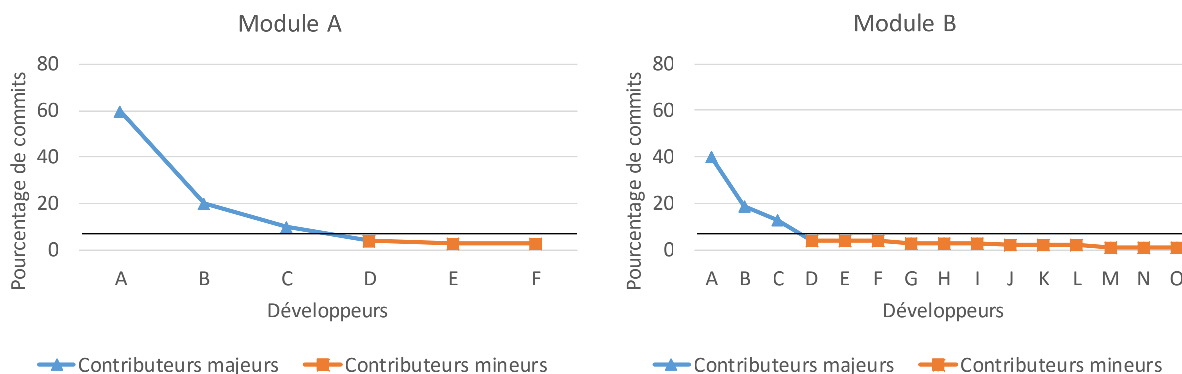

FIGURE 1.2 : Répartition des contributions dans deux modules logiciels fictifs.

Cette répartition des contribution est capturée par plusieurs métriques, dont les métriques de propriété du code définies par *Bird et al.* [Bird *et al.*, 2011] et la métrique de concentration de l'activité MAF définie par Posnett *et al.* [Posnett *et al.*, 2013]. La relation entre ces métriques et la fiabilité a été étudiée dans un nombre limité de projets : les pro-

jets développés chez Microsoft pour les métriques de propriété du code et des projets de la fondation Apache pour la métrique MAF.

Notre problématique est donc de valider l'impact de ces aspects de la répartition des contributions sur la fiabilité, mais aussi de déterminer l'importance de ces métriques lorsqu'il s'agit de prédire la fiabilité logicielle.

Dans le cas où l'effet de la répartition des contributions sur la fiabilité logicielle est avéré, les conséquences de ces résultats peuvent être abordées avec deux points de vue différents par le gestionnaire d'un projet *open-source*. Une première solution serait, dans la mesure du possible, de modifier la répartition des contributions des développeurs de manière à obtenir une configuration produisant une plus faible quantité de bogues. Cependant, il n'est pas toujours possible pour le gestionnaire d'un projet de contrôler le comportement des développeurs. Dans ce cas, il est possible d'utiliser les métriques logicielles comme indicateurs de fiabilité, afin de concentrer la maintenance préventive du projet et les sessions de relecture de code sur les modules logiciels qui sont potentiellement les plus défaillants.

Nos contributions à cette problématique sont les suivantes [Foucault *et al.*, 2014, 2015b] :

- une évaluation de la corrélation entre chacune des métriques et les mesures de fiabilité que sont le nombre et la densité de bogues par module logiciel ;
- une mesure de l'importance relative de chacune des métriques dans un modèle multivarié, afin d'évaluer l'intérêt d'ajouter ces nouvelles mesures à un jeu de mesures existant.

1.3.3 Turnover et fiabilité logicielle

Au fil de l'évolution d'un projet, l'équipe de développeurs qui y contribuent change, avec des arrivées et départs de collaborateurs mais aussi des changements de rôle de certains collaborateurs. Ce *turnover* peut être externe ou interne comme illustré par la figure 1.3 avec quatre catégories d'acteurs de la rotation des effectifs :

- un développeur sortant externe arrête ses contributions au projet dans son ensemble ;
- un développeur sortant interne arrête ses contributions à un module logiciel, mais continue ses contributions au projet ;
- un nouvel arrivant externe commence à contribuer au projet ;
- un nouvel arrivant interne commence à contribuer à un module du projet, mais contribuait déjà au projet.

Le *turnover* a été étudié à de nombreuses reprises en sociologie des organisations et dans le domaine des plateformes collaboratives et a été le sujet de quelques études dans le domaine du génie logiciel. De nombreuses théories et observations ont été réalisées concernant les différentes composantes du *turnover* :

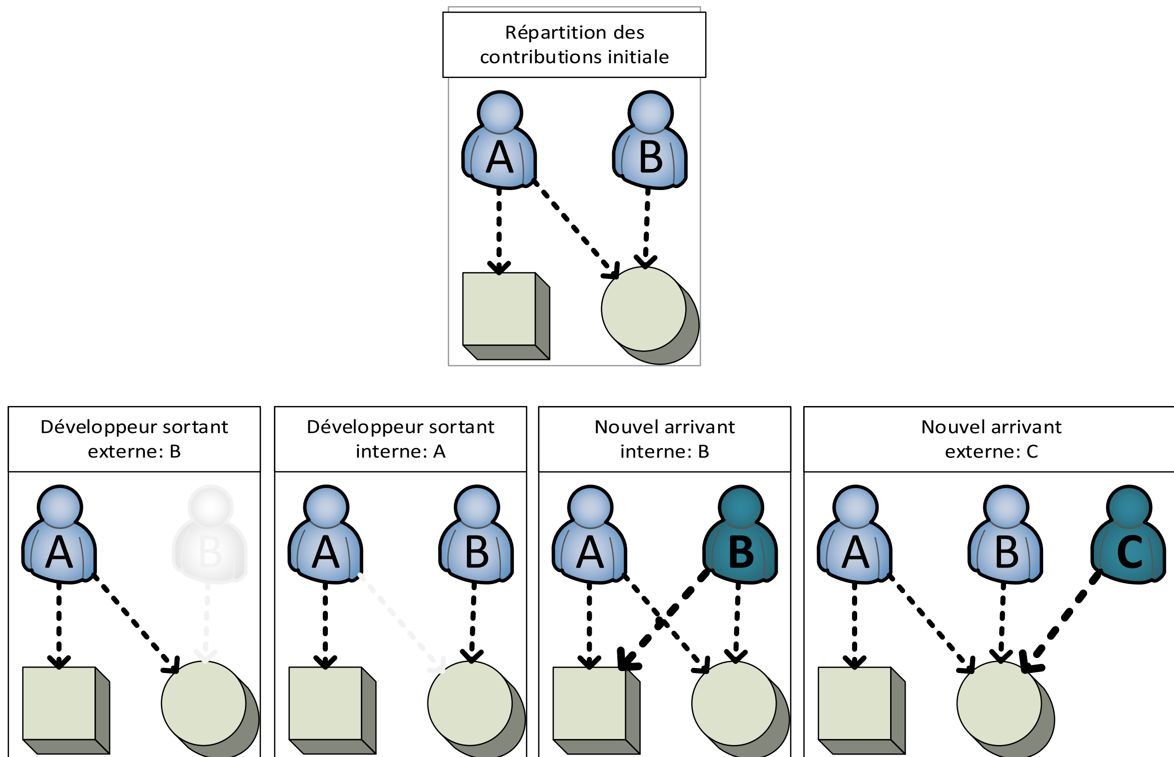

FIGURE 1.3 : Catégories de développeurs impliqués dans la rotation des effectifs.

– les départs de développeurs d'un projet entrainent une diminution des connaissances de l'équipe, ce qui tend à réduire sa productivité et la qualité de son travail [Mockus, 2010] ;
– les membres externes à un projet sont inexpérimentés lors de leur arrivée et sont plus susceptibles de commettre des erreurs [Rahman et Devanbu, 2011] ;
– les réorganisations internes permettent d'avoir des membres plus polyvalents dans une équipe et évitent la perte de motivation qui peut être observée lorsqu'une personne se voit assigner la même tâche de façon continue [Kanter, 1976].

Dans le contexte du développement logiciel, l'évolution des membres d'une équipe a été peu étudiée, et à notre connaissance aucune étude ne s'intéresse à l'impact du *turnover* sur la fiabilité des projets *open-source*. Cette problématique étant inédite dans ce contexte, il est nécessaire de définir de nouvelles métriques et décrire les schémas de *turnover* rencontrés.

Les perspectives liées à cette problématique sont similaires à celles de notre première problématique : dans le cas où une relation est observée entre *turnover* et fiabilité logicielle, il serait possible pour les gestionnaires de projets d'utiliser les métriques comme

indicateurs de fiabilité afin d'orienter la maintenance préventive vers les modules potentiellement défectueux, voire de contrôler la rotation des effectifs.

Nos contributions à cette problématique sont les suivantes [Foucault *et al.*, 2015a] :

- nous définissons les différents acteurs du *turnover* et proposons des métriques logicielles permettant la quantification de leur impact sur le code d'un projet *open-source* ;
- afin de mieux comprendre l'ampleur de la rotation des effectifs dans ce contexte, nous explorons l'évolution de l'équipe de développeurs contribuant aux projets de notre corpus ;
- nous explorons les schémas de contributions suivis par les différentes acteurs du *turnover* au sein d'un projet ;
- nous validons les métriques de *turnover* que nous avons définies en mesurant leur relation avec la fiabilité des modules logiciels et nous mesurons l'importance relative de ces métriques.

1.4 Structure

Le second chapitre de ce document fournit un aperçu de l'état de l'art, afin de mieux comprendre le positionnement de nos contributions vis-à-vis des travaux précédemment réalisés. Nous abordons ensuite chacune des contributions décrites ci-dessus, en commençant par la description de notre méthodologie, puis en consacrant un chapitre à chacune de nos problématiques. Enfin, nous apportons une conclusion générale aux travaux réalisés dans cette thèse, et abordons les perspectives de ceux-ci.

CHAPITRE

2 État de l'art

Le premier ensemble de travaux que nous étudions dans ce chapitre est lié à la façon dont est mesurée la fiabilité logicielle à partir des informations contenues dans les dépôts logiciels.

Ensuite nous abordons les travaux liés au premier aspect de l'organisation des développeurs : la répartition des contributions. Nous présentons les méthodes permettant l'extraction de ces contributions et les nombreuses mesures définies au fil des années, allant du nombre de développeurs, à des mesures plus complexes basées sur les graphes de contributions.

Le second aspect de l'organisation des développeurs est lié à l'évolution des auteurs du code source, mesurée grâce au turnover*, dont les tenants et aboutissants ont été abordés dans de nombreux domaines de recherche, avec des résultats que nous évoquons ici.*

Sommaire

2.1 Mesure de la fiabilité logicielle **12**
2.2 Répartition des contributions **14**
2.3 *Turnover* **21**
2.4 Synthèse de l'état de l'art **24**

2.1 Mesure de la fiabilité logicielle

Les problématiques de cette thèse sont toutes deux liées à la fiabilité logicielle. La manière de la mesurer a suscité de nombreux débats lors des premières conférences en génie logiciel [Littlewood, 1978]. Pour autant aujourd'hui le consensus est d'utiliser des métriques liées aux bogues identifiés pour une version donnée d'un logiciel [Hall *et al.*, 2012]. Il subsiste néanmoins de nombreuses différences, que nous présentons dans cette section, quant au procédé utilisé pour extraire ces bogues et aux informations liées aux bogues qui sont utilisées pour la mesure de la fiabilité.

2.1.1 Extraction des bogues et des *commits* correctifs

De nombreux projets utilisent un système de suivi de bogues (*bug tracker* en anglais), qui permet d'enregistrer la description d'un bogue et de planifier et suivre sa correction. Une des propriétés les plus importantes des bogues, lorsque l'on veut mesurer la fiabilité logicielle, est leur localisation dans le code source, qui peut être obtenue via les *commits* correctifs permettant de les résoudre. Le *commit* est l'objet créé dans un système de gestion de versions (ou VCS en anglais, pour *Version Control System*) lorsqu'un développeur souhaite enregistrer les modifications qu'il ou elle a apporté. Nous présentons ici les différentes approches permettant d'identifier les *commits* correctifs.

Pour faire le lien entre un bogue et les *commits* permettant de le corriger, Fischer *et al.* se basent sur le contenu des messages de *commits* ajoutés par les développeurs dans le système de gestion de versions du projet [Fischer *et al.*, 2003]. Si le message d'un *commit* contient l'identifiant d'un bogue, par exemple *« correction du bogue #1234 »*, celui-ci est considéré comme un *commit* correctif du bogue en question. Śliwerski *et al.* ont raffiné la méthode pour lier les *commits* aux bogues en ne gardant que les liens ayant un certain niveau de confiance [Śliwerski *et al.*, 2005]. Le niveau de confiance accordé à un lien augmente si, par exemple, l'auteur d'un *commit* lié au bogue est explicitement assigné à celui-ci dans le système de suivi de bogues ou si le *commit* en question est référencé dans les commentaires du bogue. Cet algorithme a été plus tard amélioré en prenant en compte la syntaxe du code (en ignorant les lignes blanches, les commentaires, le formatage du code, etc.) [Kim *et al.*, 2006].

La précision de ces algorithmes repose sur la rigueur des développeurs : si le message d'un *commit* correctif ne contient pas d'identifiant de bogue, le lien entre *commit* et bogue ne sera pas fait. Bird *et al.* ont évalué l'impact de ces liens manquants, et ont confirmé qu'ils introduisent un biais important dans les études empiriques utilisant les liens entre bogues et *commits* [Bird *et al.*, 2009a]. Wu *et al.* ont proposé un algorithme, RELINK, permettant de retrouver automatiquement ces liens manquants en se basant par exemple sur les correspondances entre les auteurs et dates des *commits* et ceux des commentaires faits sur des bogues [Wu *et al.*, 2011]. Cependant, Bissyandé *et al.* ont observé que bien que la précision

de cet algorithme soit satisfaisante (au delà de 90% de vrais positifs), son rappel ne l'est pas (seulement 10% des liens sont trouvés) [Bissyandé *et al.*, 2013].

De plus, les mesures de fiabilité basées sur les bogues enregistrées dans le système de suivi de bogues souffrent de plusieurs limitations. Premièrement, comme observé par Bachmann *et al.*, les bogues sont souvent corrigés sans qu'une entrée ne soit ajoutée dans le système de suivi de bogues [Bachmann *et al.*, 2010]. Deuxièmement, les entrées du système de suivi de bogues ne sont pas toujours classés dans la bonne catégorie, comme observé par Herzig *et al.* [Herzig *et al.*, 2013]. Dans les systèmes observées, 33,8% des entrées de bogues étaient en fait liées à de nouvelles fonctionnalités, des mises à jour de la documentation ou du *refactoring* de code. Une des conséquences de ce fait est que, en moyenne, 39% des fichiers marqués comme ayant un bogue n'en ont en fait jamais eu. Enfin, les techniques présentées ci-dessus supposent que si un fichier est modifié dans un *commit* correctif, celui-ci contient un bogue. Cependant, de nombreuses modifications du code source ne sont pas essentielles à la correction du bogue ou sont des conséquences des modifications faites dans le code défectueux, comme par exemple la propagation du renommage d'une méthode [Kawrykow et Robillard, 2011]. Kochar *et al.* ont évalué l'impact de ce biais et ont montré que bien que 28% des fichiers contenus dans des commits correctifs ne contiennent pas de bogue, cela n'a qu'un effet mineur sur le résultats de modèles visant à prédire la localisation des bogues [Kochhar *et al.*, 2014].

Tian *et al.* ont développé une technique permettant de détecter des *commits* correctifs en s'affranchissant du système de suivi de bogues [Tian *et al.*, 2012]. Cette technique s'appuie une analyse des messages de *commits* et des modifications apportées par ceux-ci dans le but d'identifier les caractéristiques spécifiques aux *commits* correctifs. Cependant, bien que le rappel de cette approche soit élevé dans le cas d'étude considéré (90%), sa précision reste assez faible (50%) et requiert une analyse manuelle des résultats. Une technique utilisée par Just *et al.* pour identifier les *commits* correctifs correspondant à de « vrais » bogues consiste à rejouer tous les tests contenus dans le dépôt de code source avant et après chaque *commit* correctif possible, et à ne conserver que les *commits* permettant de reproduire un bogue [Just *et al.*, 2014]. Bien que cette approche permette d'obtenir un jeu de données avec une précision à priori parfaite, son rappel est fortement lié au niveau de couverture du jeu de tests de chaque projet.

2.1.2 Informations additionnelles sur les bogues

Une information recherchée dans de nombreuses études est la source des bogues, c'est-à-dire le ou les *commits* ayant introduit un bogue dans le code. Afin d'identifier ces *commits* fautifs, l'algorithme défini par Śliwerski *et al.* utilise des outils similaires à `git blame`, qui permet de déterminer l'origine d'une ligne de code [Śliwerski *et al.*, 2005]. Cet algorithme, souvent référencé comme l'algorithme SZZ (pour Śliwerski, Zimmermann et Zeller, les auteurs de l'article), consiste à retrouver les *commits* qui sont à l'origine des lignes de code fautives, c'est-à-dire celles qui sont modifiées dans un *commit* correctif.

Ces *commits* sont ensuite filtrés en supprimant ceux créés après le rapport de bogue (le bogue existant déjà, les modifications faites après ne peuvent pas l'avoir introduit).

Dans certains *bug trackers*, les bogues peuvent avoir un champ correspondant à leur sévérité (on aura par exemple des bogues mineurs, majeurs ou bloquants). Certaines études utilisent ces informations liées à la sévérité des bogues, en partant du principe que des bogues plus sévères diminuent la fiabilité de manière plus importante que ceux qui sont triviaux [Shatnawi et Li, 2008]. La sévérité d'un bogue est cependant peu utilisée du fait que celle-ci est définie par les développeurs du projet, ce qui peut mener à des incohérences entre les niveaux de sévérité, dû à des différences d'interprétation entre les développeurs ou des choix politiques dans certaines équipes, comme l'ont observé Ostrand *et al.* [Ostrand *et al.*, 2005].

Mesure de la fiabilité logicielle
Nous considérons que les bogues corrigés présents dans une version d'un logiciel reflètent la fiabilité de celui-ci. Pour extraire cette fiabilité, il est nécessaire d'identifier les bogues présents dans la version du logiciel étudiée et d'en déduire les parties du logiciel défectueuses, en reconstruisant le lien en bogues et *commits*. La limitation principale des procédés d'extraction automatique existants est qu'ils supposent que les développeurs font une utilisation rigoureuse et correcte des dépôts logiciels, ce qui n'est pas toujours le cas.

2.2 Répartition des contributions

Dans cette section, nous passons en revue les travaux qui ont mesuré la répartition des contributions des développeurs.

Premièrement, nous passons en revue les différentes techniques permettant d'extraire les contributions des développeurs et les différentes limitations de ces techniques. Nous présentons ensuite les travaux utilisant la métrique « basique » liée à la répartition des contributions : le nombre de développeurs ayant contribué à un module. Ensuite, nous nous intéressons aux réseaux de contributions, qui sont des graphes représentant les contributions des différents développeurs sur les différents modules d'un logiciel, et aux métriques se basant sur ces graphes. Et enfin, nous nous intéressons aux travaux mesurant la proportion de contributions ajoutée par chaque développeur d'un module, parfois appelée propriété du code source (*code ownership* en anglais).

2.2.1 Mesurer les contributions des développeurs

Identifier le ou les auteurs d'une portion de code est une problématique couramment rencontrée par les chercheurs en génie logiciel, que ce soit dans le but de mesurer la quan-

tité de développeurs ayant travaillé sur cette portion de code [Weyuker *et al.*, 2008], de mesurer l'expertise des développeurs [Teyton *et al.*, 2013] ou d'identifier les ayants droit d'un logiciel *open-source* [Di Penta et German, 2009]. Nous passons en revue les différentes techniques permettant d'assurer une identification précise de ces contributions, en nous intéressant premièrement à l'identification de l'auteur d'un *commit*, puis à l'identification des contributeurs d'une portion de code source et à la mesure de leur niveau de contribution.

Identifier l'auteur d'un *commit*

Selon le type de gestionnaire de version (ou VCS, en anglais) utilisé par un projet, le processus de création des *commits* est différent. Dans le cas des VCS centralisés, tels que *Subversion* [1], seuls les développeurs ayant un accès en écriture sur le serveur hébergeant le VCS ont la capacité de créer ces *commits*. Les autres développeurs, s'ils souhaitent apporter des modifications au code source, doivent soumettre ces modifications via une liste de diffusion, en envoyant un *patch* qui sera relu, puis appliqué par un des développeurs ayant les droits en écriture sur le VCS [Asundi et Jayant, 2007]. Dans le cas de VCS décentralisés, tels que *Git* [2], le procédé est différent : chaque développeur a la possibilité de cloner le dépôt de code source original, puis d'ajouter des *commits* dans son clone de ce dépôt. La soumission de ces modifications dans le dépôt central se fait via l'envoi d'une demande appelée *pull request*. La réponse à cette *pull request* sera faite par un développeur ayant les droits en écriture sur le dépôt central et se matérialisera par le téléchargement des modifications du contributeur depuis son clone vers le dépôt central.

Auteur et *committer*. Le principal impact du choix du VCS sur l'identification de l'auteur d'un *commit* est le fait qu'un VCS décentralisé comme *Git* supporte nativement la distinction entre l'auteur original d'un *commit* et la personne qui a ajouté ce *commit* dans le dépôt central (appelée *committer*) [Bird *et al.*, 2009b]. Dans le cas de Subversion, seule l'identité de la personne ayant appliqué le correctif est enregistrée par le VCS. Une pratique courante dans certains projets *open-source* est d'indiquer, dans le message du *commit*, l'identité de l'auteur du correctif. Dans ce cas il devient possible de retrouver l'auteur original de façon automatique grâce à une analyse syntaxique de ces messages de *commits* [Bird *et al.*, 2007]. La précision et à fortiori le rappel de cette technique sont fortement liés à l'utilisation stricte de conventions dans les messages de *commits*, ce qui n'est pas garanti dans tous les dépôts *open-source*. Il convient de noter que le contenu d'un *commit* peut être produit par plusieurs développeurs, collaborant par exemple dans le cas de projets hébergés sur la plateforme GitHub sur une même *pull request* [Kalliamvakou *et al.*, 2014]. Dans ce cas l'information enregistrée dans le système de gestion de versions peut ne pas être suffisante pour identifier tous les auteurs d'un *commit*.

1. https://subversion.apache.org/
2. http://git-scm.com/

Alias multiples. L'identité des développeurs est enregistrée dans les gestionnaires de versions grâce à un alias composé de leur nom (ou pseudonyme) et de leur adresse de messagerie électronique. Or, il est possible qu'une même personne ait plusieurs alias entre plusieurs dépôts ou dans le même dépôt logiciel, par exemple dû au fait qu'un développeur travaille sur plusieurs machines et que la configuration du programme utilisé pour communiquer avec le VCS ne soit pas identique selon la machine. Il devient alors nécessaire d'appliquer un algorithme de fusion d'identités pour regrouper de façon précise les contributions d'un même développeur. Goeminne et Mens ont réalisé une étude comparative de différents algorithmes de fusion d'identités [Goeminne et Mens, 2013], auxquels se rajoute un algorithme développé par Kouters *et al.* [Kouters *et al.*, 2012]. Bien qu'aucun des algorithmes présentés dans ces études ne soit parfait en termes de précision et rappel, Goeminne et Mens conseillent de favoriser un algorithme avec un rappel élevé au détriment de la précision et de vérifier manuellement les faux positifs générés par l'algorithme. L'algorithme qui permettrait d'obtenir les meilleurs résultats selon cette approche est relativement simple : si deux alias ont un label (c'est-à-dire un nom, pseudonyme ou adresse de messagerie) en commun ayant une longueur dépassant un seuil donné, ces alias correspondent au même développeur et doivent être fusionnées.

Mesurer le niveau de contribution

Nous passons en revue ici les différentes techniques et définitions utilisées dans la littérature pour déterminer le ou les auteurs d'une portion de code source et pour mesurer le niveau de contribution de chacun d'entre eux.

Une première définition est de considérer qu'un développeur est le propriétaire d'une ligne de code s'il est le dernier à l'avoir modifiée. Par extension, un développeur est considéré comme propriétaire d'un fichier de code source à une version donnée s'il est le propriétaire de la majorité (relative) des lignes de code du fichier [Girba *et al.*, 2005]. Cette définition, qui est implémentée dans des outils tels que `git blame` (inclus dans le gestionnaire de versions *Git*), est restrictive par le fait qu'elle ne prend en compte que la dernière modification faite sur chaque ligne de code. Une version étendue de cette définition, proposée par Meng et al., prend en compte l'ensemble des modifications permettant de produire chaque ligne de code [Meng *et al.*, 2013]. Une autre approche pour déterminer les auteurs d'un fichier de code source consiste à prendre en compte toutes les modifications affectant le fichier, soit en considérant de manière égale toutes les modifications faites sur une période donnée [Hattori et Lanza, 2009], soit en utilisant un facteur d'oubli qui accorde un poids moins important aux modifications les plus anciennes [Hattori *et al.*, 2012].

Au delà de l'identité des auteurs du code source, certaines études utilisent la quantité de contributions produite par chaque développeur [Bird *et al.*, 2011]. Le niveau de contribution d'un développeur d peut être défini pour une ligne de code source comme le nombre de caractères contenus dans cette ligne introduits par d [Meng *et al.*, 2013]. Il est également possible de mesurer, pour un fichier donné, le nombre de *commits* effectué par

chaque développeur affectant le fichier. Cette mesure peut être raffinée en comptant, pour chaque *commit* affectant un fichier, le nombre de lignes de ce fichier affectées par le *commit* (c'est-à-dire la somme des lignes ajoutées ou supprimées), aussi appelé *churn* [Munson et Elbaum, 1998].

Ces mesures sont cependant limitées par le fait qu'elles reposent la plupart du temps sur une représentation textuelle du code. Des algorithmes de différentiation structurelle du code existent et permettent de distinguer les modifications purement esthétiques (comme le formatage ou la réorganisation du code source) des modifications ayant un impact sur le comportement du programme [Fluri *et al.*, 2007; Falleri *et al.*, 2014]. En utilisant ces algorithmes il devient alors possible de filtrer les modifications non essentielles, afin d'obtenir un calcul plus précis du niveau de contributions de chaque développeur [Kawrykow et Robillard, 2011]. D'après Kawrykow et Robillard, entre 2.8% et 22.9% (selon le logiciel étudié) des lignes de codes modifiées le seraient uniquement via des modifications qualifiées de non essentielles. Les modifications prises en compte dans cette étude représentent des ensembles de 2500 à 4200 *commits*, effectués sur des périodes allant de 240 à 2400 jours, selon le logiciel étudié [Kawrykow et Robillard, 2011].

2.2.2 Nombre de développeurs et fiabilité logicielle

La relation entre le nombre de développeurs et la fiabilité logicielle a été étudiée dès les années 70 par Brooks, avec un rapport d'expérience de développement de projets chez IBM [Brooks, 1975]. La contribution de cette étude la plus souvent citée est la "loi de Brooks", formulée comme suit : *« Ajouter des personnes à un projet en retard accroît son retard »*. L'ajout de nouveaux développeurs à un projet entraine des coûts supplémentaires, liés à la nécessité de former ces nouveaux développeurs et au fait que la synchronisation entre les développeurs devient plus complexe.

Les premiers résultats évaluant la relation entre le nombre de développeurs et la fiabilité furent mitigés. Mockus et Weiss ont construit un modèle multivarié ayant pour variable dépendante la probabilité qu'un changement du code source soit défectueux [Mockus et Weiss, 2000]. Un changement du code source est ici un ensemble de *commits* liés à une fonctionnalité donnée, et les métriques utilisées dans le modèle statistique évaluent, entre autres, le nombre de développeurs impliqués dans les *commits* et le nombre de *commits* effectués pour réaliser le changement. Une des conclusions de cette étude est que le nombre de développeurs n'a pas dans ce cas de relation significative avec la probabilité de fautes, mais que plus le nombre de *commits* nécessaires à la réalisation d'une fonctionnalité augmente, plus la probabilité de fautes augmente. Un résultat similaire a été trouvé par Graves et al., qui ont comparé l'importance de métriques de code et de procédé afin de prédire le nombre de bogues d'un module logiciel [Graves *et al.*, 2000]. Le modèle de prédiction le plus précis produit par cette étude inclut le nombre de *commits* effectués sur un module, mais, comme pour l'étude de Mockus et Weiss, ils n'ont pas trouvé de preuve de l'impact du nombre de développeurs sur la fiabilité.

Ces résultats négatifs sont une explication possible pour le peu d'études empiriques utilisant le nombre de développeurs comme indicateur de fiabilité les années suivantes. À partir de 2006 cependant, des résultats différents sont apparus : Schröter *et al.* ont observé une corrélation entre le nombre de bogues et le nombre de développeurs ou de *commits* dans le code source du projet ECLIPSE [Schröter *et al.*, 2006]. Weyuker *et al.* ont observé que l'ajout de métriques liées aux développeurs augmente de manière significative la précision de modèles visant à prédire les fichiers contenant le plus de bogues [Weyuker *et al.*, 2007, 2008]. Les métriques utilisées ici sont le nombre de développeurs dans la dernière version du projet, le nombre cumulatif de développeurs depuis le début, et le nombre de nouveaux développeurs ayant participé à la dernière version. Illes-Seifert et Paech ont mesuré, dans neuf projets open-source, la relation entre plusieurs métriques liées à l'historique de développement des fichiers [Illes-Seifert et Paech, 2008, 2010]. Parmi ces métriques, le nombre de développeurs a montré une forte corrélation avec le nombre de bogues et le nombre de *commits*.

2.2.3 Réseaux de contributions

Afin d'observer l'impact de l'augmentation du coût de la synchronisation entre les développeurs décrite par Brooks, un certain nombre d'études utilise des graphes pour représenter les contributions des développeurs sur les modules logiciels et utilisent des métriques liées à ces graphes pour évaluer la relation entre l'organisation des développeurs et la fiabilité logicielle. Capiluppi et Adams ont réalisé une évaluation empirique de la loi de Brooks sur l'écosystème de projets logiciels *open-source* KDE, en considérant les fichiers de code source ayant plusieurs contributeurs [Capiluppi et Adams, 2009]. Bien qu'une augmentation du nombre de canaux de communication, c'est-à-dire de fichiers où des développeurs doivent se synchroniser, soit observée, cette augmentation ne suit la loi de Brooks que pour un nombre de développeurs inférieur à 10. Au delà, la réorganisation du projet et des équipes de développement a permis de garder le nombre de canaux de communications relativement constant.

Cependant, ces canaux de communication ne sont pas uniformes : tous les développeurs ne travaillent pas sur le même nombre de modules, et tous les modules ne sont pas modifiés par le même nombre de développeurs. La question de la relation entre cette répartition des canaux de communication et la fiabilité logicielle se pose alors : s'il existe une configuration permettant de produire un logiciel de meilleure fiabilité, alors celle-ci devrait être favorisée par les équipes de développement. Cataldo *et al.* ont analysé deux types de dépendances de travail entre développeurs [Cataldo *et al.*, 2009]. Premièrement, les dépendances de flux de travaux, qui sont les cas où plusieurs développeurs doivent collaborer pour compléter une même tâche. Deuxièmement, les dépendances de coordination, où deux développeurs travaillant chacun sur une tâche doivent modifier les mêmes fichiers ou des fichiers syntaxiquement ou logiquement interdépendants, et doivent se coordonner pour conserver le code dans un état cohérent. Les résultats de l'étude menée par

Cataldo *et al.* sur deux projets commerciaux montrent que ces deux types de dépendances entre les développeurs augmente de façon significative le risque de bogue.

Dans de nombreux travaux, les contributions des développeurs sont représentées par un réseau de contributions qui est un graphe biparti dont une des partitions représente l'ensemble des développeurs, la seconde représente l'ensemble des modules d'un projet, et les arêtes du graphe représentent les contributions des développeurs sur les modules. En utilisant diverses métriques évaluant le degré et le couplage des différents nœuds du réseau, Pinzger et al. ont montré que plus un module est central dans le réseau de contributions, plus il aura de bogues [Pinzger *et al.*, 2008]. Il apparait également que plus un module est modifié par des développeurs répartissant leurs contributions sur un grand nombre de module, plus ceux-ci sont vulnérables aux failles de sécurité [Meneely et Williams, 2009]. Dans la même étude, Meneely et Willams ont montré que les fichiers permettant de joindre des groupes de développeurs n'ayant aucun autre canal de communication en commun sont également plus sujets aux failles de sécurité. Enfin, Posnett *et al.* ont mesuré la concentration de l'activité des développeurs sur les modules logiciels, en utilisant des mesures basées sur l'entropie [Posnett *et al.*, 2013; Blüthgen *et al.*, 2006]. Ces mesures permettent de prendre en compte à la fois le degré de concentration des contributions d'un développeur donné, avec une métrique appelée *developer activity focus* (ou DAF), et le degré de concentration des développeurs contribuant à un module donné, avec une métrique appelée *module activity focus* (ou MAF). Les résultats obtenus avec la métrique DAF montrent que plus un développeur est concentré sur un nombre restreint de modules, moins il introduira de bogues. En revanche, les résultats obtenus avec la métrique MAF montrent que les fichiers où la concentration des contributions est plus importante ont plus de bogues. Les auteurs de l'étude suspectent que ce résultat est dû à des facteurs tiers non pris en compte, comme la complexité des modules étudiés.

2.2.4 Propriété du code source

Un aspect de la répartition des contributions observé dans de nombreuses études est la propriété du code source, avec pour objectif d'identifier quels sont les développeurs responsables du code source d'un module et dans quelles proportions ceux-ci ont contribué au développement.

Mockus *et al.* ont réalisé une étude de cas sur deux projets de logiciels *open-source* : APACHE et MOZILLA [Mockus *et al.*, 2002]. Dans ces deux projets, ils ont observé la quantité de contributions réalisées par chaque développeur et ont également observé si le code de chaque module logiciel était la propriété d'un ou plusieurs développeurs. Dans le projet APACHE, la répartition du nombre de contributions par développeur est très inégale, avec 83% des *commits* réalisés par seulement 15 développeurs, sur plus de 250 développeurs ayant fait au moins une contribution. Il n'y a pas ici de forte propriété du code : dans chaque module (qui sont des fichiers ".c" dans ce cas), au moins deux développeurs ont réalisé chacun plus de 10% des contributions. Dans le projet MOZILLA en revanche, une

propriété forte du code source est imposée par le procédé de développement du projet. À chaque module logiciel est associé un développeur propriétaire, qui se charge de gérer le système de suivi de bogues et d'accepter ou non les *patchs* soumis par les autres développeurs.

Girba *et al.* ont proposé une visualisation des contributions des développeurs en proposant une « carte de propriété » (*Ownership Map*), qui peut être vue comme un cadastre du code source [Girba *et al.*, 2005]. Dans cette visualisation, un développeur est considéré comme propriétaire d'une ligne de code s'il est le dernier à l'avoir modifiée. Par extension, un développeur est considéré comme propriétaire d'un fichier de code source à une version donnée s'il est le propriétaire de la majorité (relative) des lignes de code du fichier. Cette visualisation a pour objectif l'identification de certains schémas de collaboration entre les développeurs, comme par exemple les phases de monologue d'un seul contributeur, les phases où un contributeur devient propriétaire d'un ensemble de fichiers, ou les phases où deux contributeurs collaborent sur les mêmes fichiers. Hattori et Lanza ont développé une visualisation similaire, avec pour différence que le propriétaire d'un fichier est le développeur ayant réalisé le plus de changements sur ce ficher [Hattori et Lanza, 2009]. Dans cette étude les changements sont mesurés *via* une extension de l'EDI [3] ECLIPSE. Dans une version étendue de cette visualisation, Hattori *et al.* ont raffiné le calcul de la propriété du code, en ajoutant un effet d'oubli, c'est-à-dire qu'ils considèrent que plus une modification a été faite il y a longtemps par un développeur, moins elle est importante pour mesurer la propriété du code [Hattori *et al.*, 2012]. Une autre visualisation développée par Greevy *et al.* permet de déterminer qui sont les développeurs responsables d'un module d'un projet ou d'un fonctionnalité donnée [Greevy *et al.*, 2007].

Rahman et Devanbu ont montré que les lignes de code impliquées dans un bogue sont le plus souvent le fruit d'un seul développeur, et que l'expérience d'un développeur sur un fichier en particulier a plus d'impact sur la fiabilité du code que l'expérience générale du développeur sur le projet [Rahman et Devanbu, 2011].

Bird et al. ont introduit la notion de contributeurs majeurs et mineurs, étant respectivement les développeurs ayant réalisé plus et moins de 5% des contributions d'un module logiciel [Bird *et al.*, 2011]. Dans une étude empirique menée sur l'historique de développement de Windows Vista et Windows 7, ils ont montré que les métriques de propriété de code que sont le nombre de développeurs « majeurs », le nombre de développeurs « mineurs » et le pourcentage de contributions ajoutées par le développeur principal d'un module, sont fortement corrélés avec le nombre de bogues. De plus, ils ont montré qu'ajouter ces métriques à un modèle multivarié augmente de façon significative la précision du modèle. Cette étude a été reproduite par Greiler *et al.*, qui confirment ces résultats sur d'autres projets développés chez Microsoft, en précisant que les métriques liées à la propriété du code ont une corrélation plus forte avec les bogues lorsque la granularité des modules considérée est plus grande [Greiler *et al.*, 2015].

3. Environnement de Développement Intégré

Répartition des contributions

L'identification précise de l'auteur d'un *commit* requiert l'utilisation d'un algorithme de fusion d'identités, et, si un VCS centralisé est utilisé, d'une analyse syntaxique des message de *commits*. Pour mesurer le niveau de contribution des développeurs, il est nécessaire d'effectuer des choix concernant la granularité des éléments de code source pris en compte.

Le type d'analyse, textuelle ou syntaxique, qui est faite du code source est également un choix important : une analyse prenant en compte la syntaxe des langages de programmation utilisés permettra d'obtenir des résultats plus précis mais a un coût plus élevé en termes de temps d'extraction et de développement des outils permettant cette extraction.

De l'ensemble d'études incluant le nombre de développeurs et le nombre de *commits* comme indicateurs de fiabilité, nous observons, premièrement, que les résultats concernant le nombre de développeurs diffèrent d'une étude à une autre, et sont probablement impactés par le contexte du projet, et deuxièmement, que le nombre de *commits* est généralement lié à la fiabilité du code : plus un code est modifié, plus il aura de bogues.

La coordination des développeurs et la concentration de leur activité sont deux aspects de la répartition des contributions sont mesurés via des réseaux de contributions dans de nombreuses études. La plupart de ces études confirment que la coordination des développeurs est un facteur influant la fiabilité logicielle.

Enfin, un autre ensemble de travaux prend en compte le niveau de contribution des développeurs afin de déterminer les propriétaires de modules logiciels (ou d'autres éléments de code source). Les conclusions de ces travaux, et en particulier ceux de Bird *et al.* et Greiler *et al.* confirment la présence d'une relation entre l'importance des propriétaires d'un module logiciel et la fiabilité de celui-ci.

2.3 *Turnover*

Les travaux présentés dans la section précédente se concentrent sur la répartition des contributions apportées par une équipe de développeurs pendant une période donnée. Or, cette équipe de développeurs évolue au cours du temps et un *turnover*, que l'on peut traduire par « rotation des effectifs » a lieu. Dans cette section nous passons brièvement en revue les travaux décrivant l'impact du *turnover* dans le domaine de la sociologie des organisations, domaine dans lequel il a été étudié de façon répétée. Puis nous nous intéressons au *turnover* dans les plateformes de collaboration en ligne, telles que Wikipédia, qui est similaire en termes de modes de communication aux projets de développement *open-source* [Capiluppi, 2013]. Et enfin nous aborderons les travaux liés au *turnover* dans le domaine du génie logiciel et à son impact sur la fiabilité logicielle.

2.3.1 Dans la sociologie des organisations

Nous nous intéressons aux travaux décrivant les effets du *turnover* dans le domaine de la sociologie des organisations car cette problématique, relativement récente et peu étudiée en génie logiciel, a été au contraire le sujet de nombreux articles de recherche dans le domaine susmentionné.

Le *turnover* a un impact sur la productivité, les relations avec les clients et les performances d'un groupe ou d'une organisation [Hausknecht et Trevor, 2011]. Cependant, de nombreuses études suggèrent que, bien qu'un *turnover* important a un impact négatif sur les aspects énoncés ci-dessus, la relation entre le *turnover* et la productivité n'est pas linéaire, et que l'absence de *turnover* n'est pas la situation optimale [Hancock *et al.*, 2013]. Cet impact sur la productivité et le profit est accentué dans les domaines qui requièrent un niveau plus élevé de connaissances, tels que le domaine des finances ou de la technologie [Hancock *et al.*, 2013]. De plus comme, observé par Ton et Huckman, l'augmentation du *turnover* n'a pas d'impact sur la performance de magasins (qui sont dans ce cas des librairies) où les procédés de travail sont standardisés, alors que la performance des magasins n'adoptant pas ces standards est fortement dégradée en présence d'un fort *turnover* [Ton et Huckman, 2008].

Le *turnover* peut également être interne à un groupe : dans la sociologie des organisations, ce *turnover* interne est mesuré avec le nombre d'employés qui changent de fonction au sein d'une organisation [Hamermesh *et al.*, 1996]. Ces actions sont motivées par des possibilités d'évolutions de carrière dans le but d'obtenir de meilleures rémunération et autonomie et de se voir confier plus de responsabilités et d'exprimer de nouvelles compétences [Thompson, 1967]. Kanter mentionne dans ses travaux le fait que sans *turnover* interne, les membres d'une équipe ont de plus faibles ambitions et sont moins impliqués dans leur travail, dû à l'absence de possibilités d'évolution [Kanter, 1976].

2.3.2 Dans les plateformes de collaboration en ligne

Le *turnover* a également été étudié dans les plateformes de collaboration en ligne. Cet environnement partage plusieurs similarités avec les projets *open-source*, telles que l'absence relative de barrières d'entrée et de sortie et la forte distribution géographique des contributeurs.

Dans la version anglaise de Wikipédia, le *turnover* est usuellement élevé, et 60% des contributeurs ne contribuent que pendant une seule journée [Panciera *et al.*, 2009]. L'hypothèse que le *turnover* affecte la qualité du travail produit de façon non-linéaire, émise en sociologie des organisations, est également supportée dans le cas de Wikipédia [Ransbotham et Kane, 2011], du moins lorsque les connaissances des nouveaux arrivants sont supérieures aux connaissances des personnes ayant quitté le groupe éditant un article. Ces nouveaux arrivants ont ici encore un impact positif sur le groupe car ils peuvent relancer certaines discussions et amener une augmentation de la participation des autres membres

du groupe [Dabbish *et al.*, 2012]. L'impact négatif entrainé par un *turnover* trop élevé est cependant toujours observé, comme l'ont noté Qin *et al.* dans le cas de projets de Wikipédia (qui « permettent de coordonner les efforts de la communauté en regroupant les contributeurs autour de thématiques. » [4])[Qin *et al.*, 2014].

2.3.3 Dans le génie logiciel

Dans le domaine du génie logiciel, le *turnover* est, de la même manière que dans les autres domaines, perçu négativement par les professionnels, comme l'ont montré Hall *et al.*, en s'appuyant sur des interviews de développeurs expérimentés, employés dans une entreprise du domaine de la finance [Hall *et al.*, 2008]. L'impact du *turnover* sur la fiabilité logicielle a été étudié par Mockus, qui, dans une étude empirique basée sur l'historique d'un projet industriel, a montré que les départs de développeurs entrainent un nombre de bogues plus élevé, dû à la perte de connaissances inhérente à ces départs. Cependant les arrivées de nouveaux développeurs ont un effet négligeable sur la qualité, une hypothèse de ce phénomène étant que les nouveaux arrivant ne se voient pas assigner de modifications importantes [Mockus, 2010]. Mockus a également étudié la succession des développeurs sortants, c'est-à-dire la reprise du code d'un développeur sortant par un nouveau développeur, et a proposé divers critères influant la productivité des développeurs lors de cette succession [Mockus, 2009]. Afin de mesurer l'impact du départ d'un développeur, Izquerdo-Cortazar considère les lignes « orphelines » du code source, c'est-à-dire les lignes dont l'auteur a quitté le projet [Izquierdo-Cortazar, 2008].

Le *turnover* dans les projets *open-source* est hautement spécifique à chaque projet, comme l'ont montré plusieurs études. Robles *et al.* ont mesuré l'évolution de l'équipe de développement du projet GIMP, et ont observé que l'équipe centrale de développeurs évolue peu pendant l'histoire du projet [Robles *et al.*, 2009]. Mens *et al.* ont observé les migrations de développeurs de l'écosystème GNOME, et ont conclu que ces migrations, y compris au sein du même écosystème, sont spécifiques à chaque projet [Mens *et al.*, 2014]. GIMP, mentionné ci dessus, est inclus dans les projets observés par Mens *et al.*, et reçoit en moyenne un nombre plus faible de nouveaux arrivants que des projets comme EVOLUTION ou GTK. Sharma *et al.* ont observé que le niveau de *turnover* des projets *open-source* pouvait être prédit en utilisant des métriques telles que le niveau d'activité et l'âge du projet [Sharma *et al.*, 2012].

4. http://fr.wikipedia.org/wiki/Projet:Accueil

Turnover
L'ensemble de travaux réalisés en sociologie des organisations nous permet d'avoir une vue d'ensemble sur le phénomène du *turnover*. Les résultats de ce études sont en faveur d'un impact non linéaire du *turnover* sur les aspects de qualité et de productivité : un *turnover* trop élevé nuit à la qualité et la productivité, tout comme une absence de *turnover*. Quant au *turnover* interne, il est majoritairement perçu comme permettant de maintenir l'implication des membres d'une organisation. Les résultats obtenus en étudiant les projets de Wikipédia sont similaires à ceux de la sociologie des organisations : le *turnover* impacte négativement la qualité des pages produites mais il est nécessaire à l'apport de nouvelles connaissances. L'impact de la rotation des effectifs dans les projets logiciels a été évalué dans un contexte industriel, où une relation entre les départs de développeurs et la fiabilité du logiciel a été observée. Dans le contexte *open-source*, des mesures de l'ampleur du *turnover* ont été effectuées, sans que la problématique de la relation avec la fiabilité ait été abordée.

2.4 Synthèse de l'état de l'art

Cet état de l'art apporte de nombreuses informations concernant la méthodologie à adopter lors de l'extraction des contributions des développeurs et des mesures de qualité depuis les dépôts de code source.

Au vu de cet état de l'art, il apparait que :

- notre première problématique, à savoir la relation entre la répartition des contributions et la qualité logicielle, a été étudiée de façon approfondie via l'usage de nombreuses métriques. Reproduire ces études sur des projets *open-source* fait partie du procédé de la recherche empirique et permettrait de confirmer ou d'améliorer les théories développées dans ces travaux;
- notre seconde problématique, liée à la rotation des effectifs dans les projets *open-source* et son impact sur la fiabilité, n'a pas été étudiée à notre connaissance. Cependant, les phénomènes de *turnover* et leur impact ont été étudiées dans d'autres domaines, tels que la sociologie des organisations et les plateformes de collaboration en ligne. Il est donc possible de réutiliser les théories établies dans ces domaines, et de vérifier leur application aux projets *open-source*.

CHAPITRE

3 Méthodologie

L'étude des problématiques de cette thèse passe par la sélection de dépôts logiciels appropriés, l'extraction des informations pertinentes de ces dépôts et l'analyse de ces informations à l'aide d'outils statistiques. Nous définissons ici une méthodologie applicable à nos deux problématiques de recherche et à toute problématique liée à la relation entre l'organisation des développeurs et la fiabilité. Cette méthodologie permet la construction d'un jeu de données précis tout en répondant à un certain nombre d'exigences présentées dans ce chapitre. Ce chapitre décrit les critères de sélection des dépôts, le procédé utilisé pour l'extraction des métriques d'organisation et les méthodes utilisées pour l'analyse statistique de celles-ci. Une partie essentielle de la recherche empirique étant la reproduction des résultats et leur réplication sur de nouveaux échantillons de données, nous mettons à disposition la suite logicielle utilisée pour tous les résultats présentés dans cette thèse. Cette suite logicielle et les jeux de données extraits sont également présentés dans ce chapitre.

Sommaire

3.1 Aperçu de la méthodologie **26**
3.2 Critères de sélection des projets **29**
3.3 Extraction des métriques **32**
3.4 Analyse statistique des métriques **38**
3.5 Réplication des résultats **43**
3.6 Validité de la méthodologie **45**

3.1 Aperçu de la méthodologie

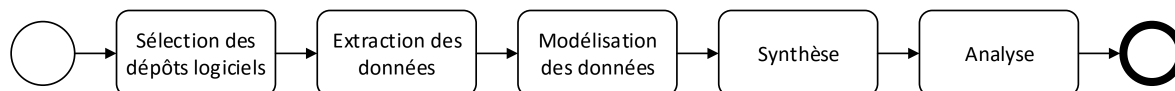

FIGURE 3.1 : Étapes du procédé de fouilles de données dans les dépôts logiciels [Hemmati *et al.*, 2013].

La méthodologie que nous utilisons pour répondre à nos problématiques fait partie du domaine de la fouille de données dans les dépôts logiciels, aussi appelé *mining software repositories* (ou MSR) en anglais. Les études de ce domaine se découpent généralement en cinq étapes, illustrées figure 3.1. Ces étapes sont : la sélection des dépôts logiciels et des projets étudiés, l'extraction des données depuis ces dépôts logiciels, la modélisation de ces données afin qu'elles représentent les concepts souhaités, la synthèse de ces données à l'aide de métriques et l'analyse de ces métriques. Enfin, les études de fouille de données dans les dépôts logiciels étant des études empiriques, la reproduction de celles-ci fait partie de la méthodologie de recherche.

Notre méthodologie se découpe en deux phases, illustrées par les diagrammes BPMN [1] des figures 3.3 et 3.4. Ces diagrammes utilisent les notations décrites dans la figure 3.2.

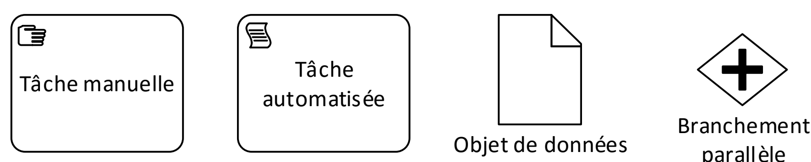

FIGURE 3.2 : Légende des diagrammes BPMN.

La première phase de notre méthodologie a pour but d'extraire les métriques d'organisation des développeurs. La figure 3.3, page 27, présente les différentes étapes de cette phase. Les deux premières étapes sont le découpage du code source en modules, réalisé manuellement, et l'identification des alias des développeurs, réalisée à l'aide d'un algorithme dont les résultats sont validés manuellement. Cette étape d'identification des alias a pour but de fusionner les alias appartenant à un même individu. Une fois ces informations extraites, il est possible de produire les différentes métriques. Les métriques de fiabilité requièrent l'extraction de l'identité des *commits* correctifs, réalisée manuellement,

1. *Business Process Model and Notation*

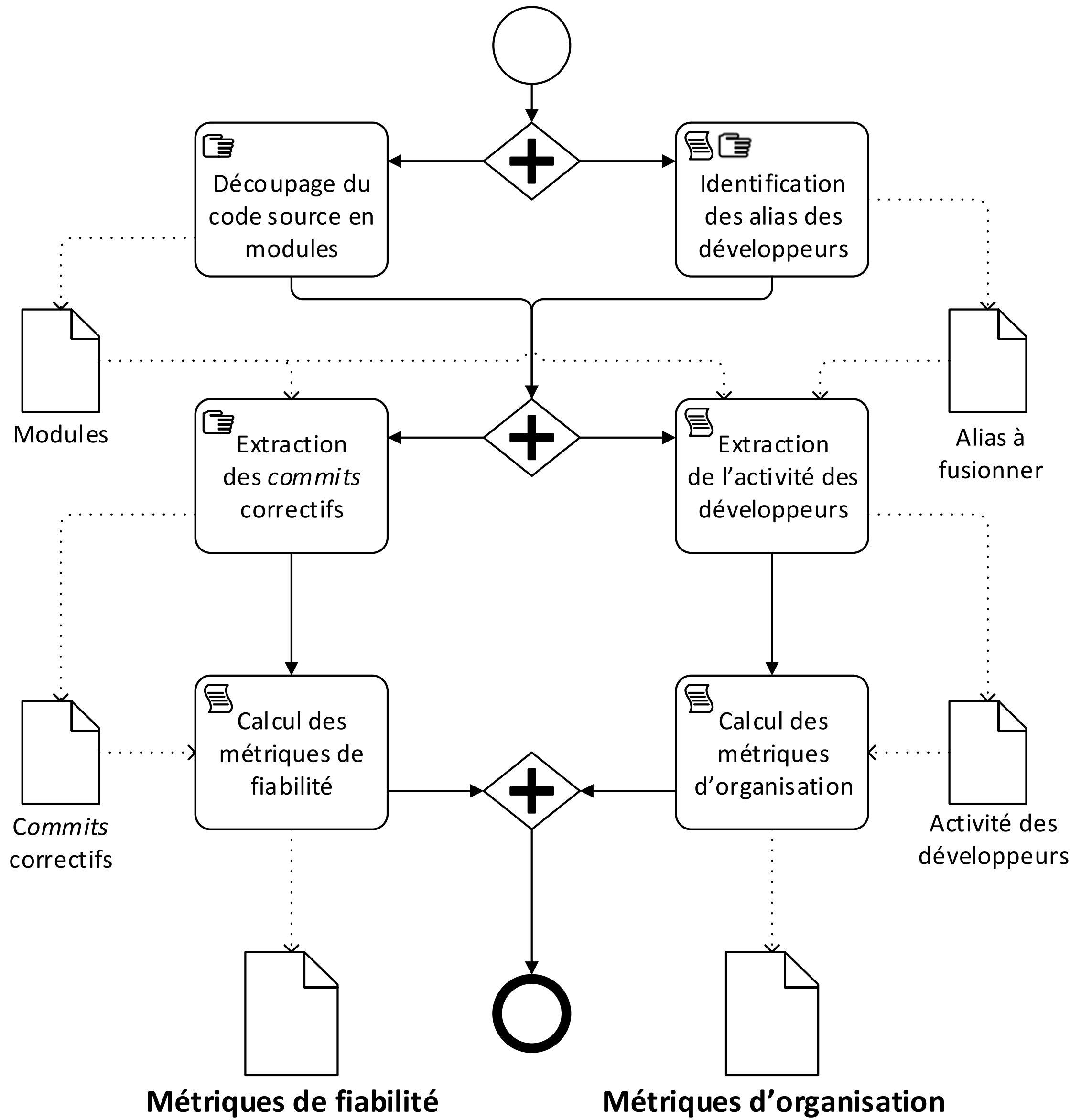

FIGURE 3.3 : Procédé d'extraction des métriques.

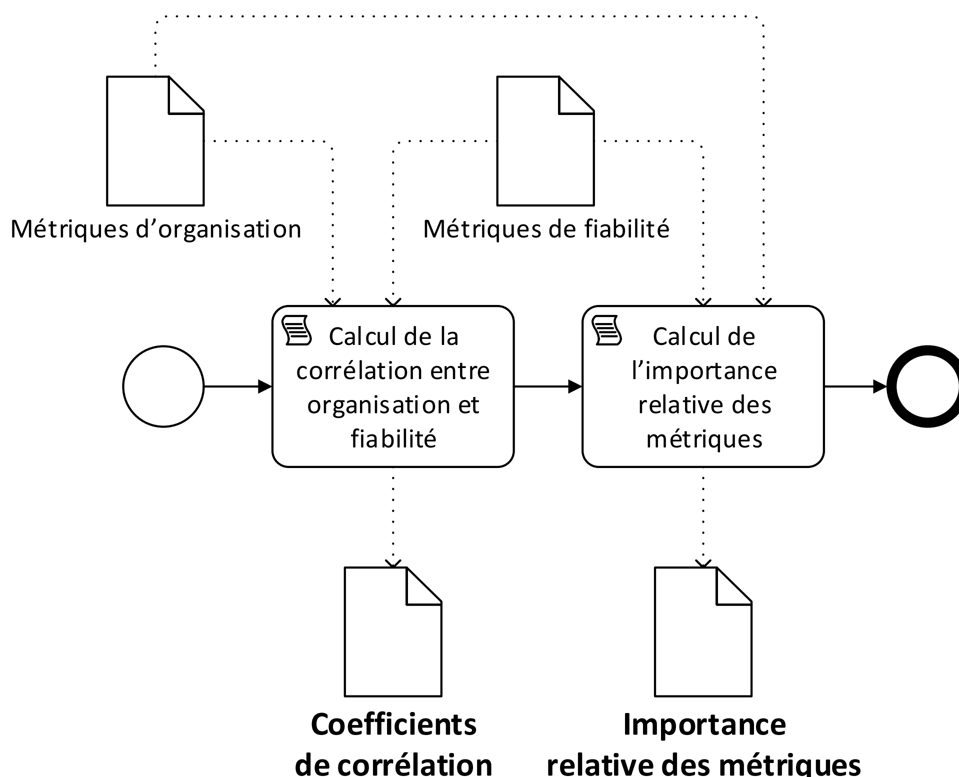

FIGURE 3.4 : Procédé d'analyse des métriques.

tandis que les métriques d'organisation des développeurs requièrent l'extraction de l'activité de ces derniers. Différents documents intermédiaires sont donc produits au cours de cette phase :

- la liste des modules du code source ;
- la liste des alias appartenant aux mêmes développeurs et devant être fusionnés ;
- la liste des *commits* correctifs ;
- l'activité des développeurs.

Nous détaillons les différences étapes de cette phase dans la section 3.3.

La seconde phase de notre méthodologie est une phase d'analyse statistique des métriques issues de la première phase (figure 3.4, page 28). Les résultats produits sont les coefficients de corrélation entre métriques d'organisation et de fiabilité ainsi que les mesures d'importance relative des métriques. Ce sont ces résultats qui nous permettront d'apporter des réponses à nos problématiques de recherche. Les différentes étapes de cette seconde phase sont détaillées dans la section 3.4

3.2 Critères de sélection des projets

Nous présentons ici les exigences liées à la sélection de projets adaptés à nos problématiques, c'est-à-dire permettant l'extraction d'informations précises concernant l'activité des développeurs et la fiabilité logicielle.

3.2.1 Gestionnaire de versions

Comme nous l'avons vu dans le chapitre précédent, tous les gestionnaires de versions ne permettent pas une extraction fiable de l'identité de l'auteur d'un *commit*.

Dans les gestionnaires de versions centralisés, tels que Subversion ou CVS, les *commits* sont enregistrés directement dans le dépôt central, par un développeur ayant la permission d'écrire sur celui-ci. Dans le cas où un développeur n'ayant pas les droits en écriture sur le dépôt central souhaite soumettre des contributions, il le fait via l'envoi d'un *patch*, qui sera alors appliqué par un des développeurs ayant les droits en écriture. Seule l'identité de ce dernier est prise en compte nativement par le gestionnaire de versions et l'identité de la personne ayant envoyé le *patch* est, dans certains projets, renseignée dans le message du *commit*. Cependant, il n'existe aucune garantie que cette pratique soit suivie rigoureusement et il est également possible que certains *commits* contiennent les modifications correspondant à plusieurs *patchs*, soumis par plusieurs développeurs, auquel cas il deviendrait impossible d'identifier les modifications de chacun des développeurs.

Critère 3.2.1

Les gestionnaires de versions décentralisés, tels que Git ou Mercurial, permettent d'identifier l'auteur de chaque commit de façon distincte de la personne qui l'a ajouté au dépôt central. Pour cette raison, les projets sélectionnés doivent utiliser un gestionnaire de versions décentralisé. Afin de trouver des projets correspondant à ce critère, nous restreignons notre espace de recherche à la plateforme d'hébergement GitHub, contenant plusieurs millions de dépôts Git (la barre des dix millions de dépôts a été atteinte en 2013 [a]).

a. https ://github.com/blog/1724-10-million-repositories

Il faut noter cependant que les dépôts hébergés sur GitHub, apparaissant comme étant des dépôts Git, peuvent être des miroirs de dépôts Subversion hébergés sur d'autres plateformes. C'est le cas notamment de la plupart des projets de la fondation APACHE. Les informations stockées dans ces dépôts sont des copies des informations stockées dans les dépôts Subversion et ne permettent pas donc l'identification précise des auteurs des *commits*. Il est donc nécessaire de vérifier lors de la sélection des projets, que les développeurs utilisent bien le gestionnaire de versions Git, ce qui peut être fait en parcourant les instructions de contributions de chaque projet, ou en analysant les messages de *commit*, qui,

dans le cas d'un miroir Git, font référence à l'identifiant du *commit* dans Subversion. De plus, il est possible qu'un même dépôt ait migré d'un gestionnaire de versions à un autre. C'est le cas de certains dépôts de la fondation ECLIPSE par exemple, dont l'historique a été migré de CVS à Subversion, puis enfin à Git.

Critère 3.2.2
Les *commits* analysés doivent avoir été créés avec Git afin de permettre l'extraction de l'identité des développeurs.

Afin de distinguer les *commits* créés en utilisant Git de ceux importés depuis un autre gestionnaire de versions, nous parcourons manuellement la liste des *commits* afin de déterminer le moment où la migration vers Git a été réalisée, le cas échéant. Lors de l'import depuis un autre gestionnaire de versions, Git ajoute une ligne de texte aux messages de *commits* importés, ce qui permet d'identifier ces derniers.

3.2.2 Architecture des branches de développement et de maintenance

Comme nous allons voir dans la section 3.3, notre procédé d'extraction de métriques consiste en partie, pour chacun des projets étudiés, à analyser manuellement les *commits* afin d'identifier ceux qui corrigent les bogues d'une version donnée. Pour faciliter cette extraction manuelle (dont le choix est motivé par l'étude des travaux existant, voir section 2.1.1), il est nécessaire de filtrer les projets sélectionnés en fonction de l'architecture de branches utilisée par leur gestionnaire de versions.

Dans un gestionnaire de versions, il est possible d'appliquer un *commit* à différentes branches de l'historique de développement afin de permettre l'évolution simultanée de différentes versions du logiciel produit. Dans certains projets, cette fonctionnalité des gestionnaires de versions est utilisée pour créer une branche de maintenance spécifique à une version v_i du logiciel, ayant pour but de garantir la stabilité et le support à long terme de cette version. Contrairement aux *commits* des branches de production qui peuvent avoir comme objectif d'ajouter de nouvelles fonctionnalités et de répondre à de nouvelles exigences, les modifications appliquées aux branches de maintenance ont pour but de :

- corriger des bogues ;
- améliorer la documentation du logiciel ;
- apporter des modifications permettant d'assurer la compatibilité avec de nouvelles versions de dépendances du logiciel, comme par exemple, dans le cas d'un logiciel développé en Ruby, des évolutions de la bibliothèque standard de ce langage de programmation.

Le schéma de la figure 3.5 illustre l'architecture de branches recherchée.

La présence d'une branche de maintenance associée à une version v_i nous garantit également que les *commits* correctifs ont pour but de corriger des bogues qui sont pré-

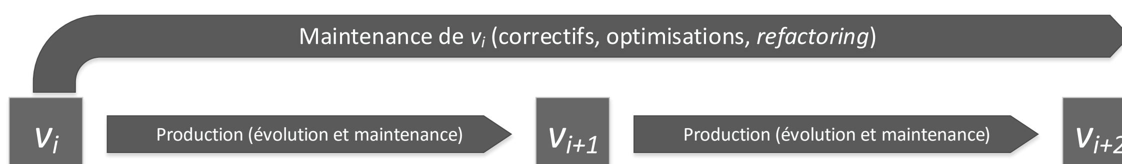

FIGURE 3.5 : Architecture de branches recherchée dans les gestionnaires de versions des projets sélectionnés.

sents dans la version v_i. Sans cette branche, les *commits* correctifs et les *commits* ajoutant de nouvelles fonctionnalités seraient appliqués en continu et il deviendrait difficile de déterminer, pour chaque commit correctif, si le bogue corrigé est présent à la version v_i ou s'il est lié à une des nouvelles fonctionnalités ajoutées depuis v_i.

Critère 3.2.3
Les projets inclus dans notre corpus doivent donc utiliser, pour au moins une version du logiciel, une branche contenant des tâches de maintenance réalisées en parallèle de la production d'une nouvelle version du logiciel.

3.2.3 Autres critères de sélection

Langage de programmation

Une problématique récurrente lors de la réalisation d'études empirique est la généralisation des résultats, aussi appelée validité externe de l'étude. Dans le cas d'études empiriques se basant sur de dépôts logiciels, un critère pouvant impacter la généralisation des résultats est le choix du langage de programmation utilisé dans les projets étudiés. Si tous les projets étudiés sont développés dans le même langage, un risque encouru est que les conclusions de l'étude ne s'appliquent qu'aux projets développés dans ce langage.

Critère 3.2.4
Le corpus de projets sélectionnés doit contenir des projets développés dans différents langages de programmation, dans de but de permettre une meilleure généralisation de nos résultats.

Taille et âge des projets

Afin de pouvoir observer les phénomènes de répartition des contributions et de rotation des effectifs décrits dans nos problématiques, il est nécessaire que les projets étudiés aient un nombre suffisant de développeurs et un historique suffisamment long. En se basant sur les observations réalisées lors de nos travaux, nous avons fixé des seuils permettant d'obtenir des projets où les phénomènes liés à nos problématiques sont observables.

Critère 3.2.5
Les projets sélectionnés doivent avoir au minimum cinquante développeurs dans leur historique de développement et la durée de cet historique doit être d'au moins deux ans.

3.3 Extraction des métriques

Le diagramme présenté figure 3.3, page 27, résume le procédé d'extraction des métriques. Afin d'obtenir un jeu de données précis, certaines tâches de ce procédé nécessitent d'être réalisées manuellement : le découpage du code source en module, l'identification des alias des développeurs et l'extraction des *commits* correctifs. Nous présentons ici le détail de ces tâches et les exigences auxquelles elles répondent.

Nous présentons également les trois dernières étapes du procédé d'extraction des métriques, à savoir l'extraction de l'activité des développeurs, le calcul des métriques de fiabilité et le calcul des métriques d'activité, qui sont des tâches automatisées.

3.3.1 Découpage du code source en modules

Les mesures réalisées dans nos contributions concernent l'activité des développeurs sur les différents modules d'un logiciel. Afin de réaliser ces mesures, il est donc nécessaire de réaliser au préalable un découpage du code source en différents modules.

Exigence 3.3.1
Le procédé de découpage du code source doit s'adapter au critère 3.2.4 et doit donc être indépendant du langage de programmation utilisé.

Pour un même programme, il est possible d'obtenir plusieurs décompositions en modules, en se basant sur les dépendances entre ses différents éléments [Mitchell et Mancoridis, 2006] ou encore sur l'évolution du code, en considérant que les fichiers changés ensemble appartiennent au même module [Beck et Diehl, 2012].

Afin de s'affranchir des problématiques de validation de ces algorithmes, nous avons choisi de réaliser cette étape manuellement. Un module défini par notre procédé peut être composé de :

- un fichier ;
- un dossier et tous les fichiers et dossiers contenus dans celui-ci ;
- un dossier et les fichiers contenus dans celui-ci, en excluant les dossiers qu'il contient, qui seront alors considérés comme d'autres modules.

Pour réduire le biais induit par la subjectivité des choix de découpage, nous avons demandé à trois juges de réaliser indépendamment un découpage en modules pour chaque projet. Les trois juges, tous des étudiants en thèse en génie logiciel, se sont ensuite concertés afin de fusionner leurs résultats. Nous avons sélectionné sept projets pour notre méthodologie, que nous présentons plus loin dans ce chapitre (section 3.5.3). Pour quatre des projets, soit AngularJS, Ansible, JQuery et Mono, les trois juges ont produit des découpages sensiblement identiques. Dans le cas de Jenkins, Rails et PHPUnit, un accord fut initialement rencontré entre deux des juges, le troisième ayant choisi une granularité plus large pour les modules. Nous avons choisi le découpage créé par la majorité.

Solution 3.3.1

Le critère de sélection de modules se base sur les noms de dossiers, fichiers et, dans le cas où ces informations sont disponibles, des fichiers de configuration du projet (tels que des fichiers *assembly* pour le langage C#), le but étant de définir des modules regroupant des fonctionnalités similaires d'un point de vue sémantique. Ce critère étant fortement subjectif, nous avons demandé à trois juges de produire un découpage en modules du code de chacun des projets à la version v_i.

3.3.2 Identification des alias des développeurs

Lors de l'évolution du projet, il est possible qu'un même contributeur enregistre ses modifications sous différents alias, dû à des modifications de la configuration de son client Git.

Exigence 3.3.2

Une même personne pouvant avoir plusieurs alias dans le gestionnaire de versions, il est nécessaire de détecter ces identités multiples afin de les fusionner lors de l'extraction des contributions des développeurs.

Notre critère principal concernant l'identification des alias multiples est que celle-ci ait un rappel aussi élevé que possible, c'est-à-dire qu'il faut maximiser les identités multiples existantes identifiées par l'algorithme. Nous utilisons l'algorithme « simple » recommandé

par Goeminne et Mens [Goeminne et Mens, 2013] : nous utilisons les champs « nom » et « adresse électronique » (dont seule la partie locale, avant le caractère @, est conservée) de chaque *commit*, qui sont ensuite normalisés. La normalisation des champs se fait de la manière suivante :

- suppression des espaces en début et fin de champ ;
- remplacement de tous les séparateurs de mots (un ou plusieurs espaces, tabulation, point, trait d'union, tiret bas) par un espace ;
- suppression des diacritiques ;
- conversion des lettres en minuscule.

Nous considérons ensuite que l'alias de chaque développeur est formé par l'ensemble des mots ayant une longueur d'au moins trois caractères. Par exemple, si un développeur enregistre un commit avec l'alias ⟨*Mr_John_Doe, jdoe@gmail.com*⟩, l'ensemble de mots constituant cet alias sera {*john, doe, jdoe*}. Nous considérons ensuite tous les alias ayant une intersection non vide comme candidats à la fusion. Si dans le même projet que pour l'exemple précédent, un *commit* a été enregistré avec l'alias ⟨*Dr_Potato, jdoe@outlook.com*⟩, l'ensemble de mots constituant cet alias sera {*jdoe, potato*} et les deux alias seront donc candidats à la fusion.

Solution 3.3.2
L'algorithme décrit ci-dessus ayant une précision relativement faible, compensée par son rappel élevé, nous vérifions manuellement chacun des résultats produits par celui-ci (qui sont des groupes d'alias à fusionner). L'automatisation complète de cette étape du procédé d'extraction n'étant pas nécessaire, cette solution est satisfaisante dans notre cas.

3.3.3 Extraction des *commits* correctifs

Exigence 3.3.3
Les mesures de fiabilité utilisées doivent refléter la quantité de bogues présents dans la version v_i de chaque projet.

Notre analyse de l'état de l'art ne nous permettant pas de trouver un algorithme permettant une extraction automatique et précise de la fiabilité logicielle, nous adoptons une approche manuelle. Étant donnés les problèmes liés à la classification erronée des entrées dans les systèmes de suivi de bogues [Herzig *et al.*, 2013] et le fait que de nombreux bogues soient corrigés sans qu'une entrée soit ajoutée dans le système de suivi de bogues du projet [Bachmann *et al.*, 2010], nous choisissons de nous passer des informations contenues dans ce type de dépôt logiciel et de nous reposer exclusivement sur les informations fournies par le système de gestion de versions de chaque projet.

La mesure de fiabilité que nous souhaitons extraire est, pour chaque module, le nombre de *commits* ayant pour but de corriger un bogue présent dans la version v_i du projet. Nous excluons donc les bogues corrigés durant la phase de production du logiciel. Comme nous l'avons mentionné dans la section précédente, les projets que nous avons sélectionnés contiennent tous une branche de maintenance associée à la version v_i.

Solution 3.3.3

Afin de mesurer la fiabilité des différents modules logiciels, nous parcourons manuellement tous les *commits* de la branche de maintenance associée à la version v_i et comptons le nombre de *commits* correctifs appliqués à chacun des modules.

Notre définition d'un *commit* correctif inclut tous les *commits* contenant des modification du code source affectant le comportement du logiciel (les *commits* modifiant des commentaires ou le formatage du code sont donc ignorés), dans le but de corriger un bogue. Les types de bogues considérés incluent :

- les erreurs logiques ou arithmétiques (par exemple une division par zéro) ;
- les bogues liés à l'allocation de ressources (par exemple un pointeur nul, un dépassement de tampon, etc.) ;
- les erreurs d'exécution parallèle comme des interblocages ou des situations de compétitions (respectivement *deadlock* et *race conditions* en anglais) ;
- les bogues d'interface (par exemple, le mauvais usage d'une interface de programmation, l'implémentation incorrecte d'un protocole, les bogues liés aux caractéristiques d'une plateforme matérielle ou d'un système d'exploitation, etc.)
- les vulnérabilités de sécurité ;
- les bogues liés à une mauvaise compréhension du cahier des charges ;
- les défauts de conception.

L'identification des *commits* correspondant aux caractéristiques ci-dessus est réalisée manuellement, en prenant soin d'ignorer les *commits* ajoutant de nouvelles fonctionnalités au programme ou réalisant des optimisations de performance, cet aspect de la qualité logicielle étant distinct de la fiabilité. Nous ignorons également les *commits* résolvant des erreurs liées à l'évolution d'une dépendance tierce, car nous considérons que ces erreurs ne sont pas liées à un défaut de fiabilité du logiciel. Enfin, dans certains cas, un *commit* correctif peut être la cause d'un nouveau bogue, ce qui entraine son annulation par les développeurs une fois ce nouveau bogue détecté, par le biais d'un nouveau *commit* défaisant les modifications du *commit* fautif. Dans ce cas, le *commit* fautif et celui qui l'annule sont tous les deux ignorés. Nous considérons qu'un *commit* correctif est lié à un seul bogue, c'est à dire que nous ne considérons pas la possibilité qu'un *commit* puisse contenir des modifications corrigeant plusieurs bogues.

3.3.4 Calcul des métriques de fiabilité

Une fois la liste des *commits* correctifs extraits, deux mesures de fiabilité sont automatiquement calculées pour chaque module :

- le nombre de *commits* correctifs affectant au moins un fichier de ce module ;
- la densité de *commits* correctifs pour le module, c'est-à-dire le nombre de *commits* correctifs divisé par la taille du module, mesurée en nombre de lignes de code (sans compter les lignes vides ou de commentaires).

3.3.5 Extraction de l'activité des développeurs

La tâche d'extraction de l'activité des développeurs consiste à mesurer la quantité de contributions réalisées par chaque développeur sur chacun des modules des projets et pendant une certaine période de temps. La longueur de cette période est un des paramètres des métriques d'activités, et son influence sur les résultats sera étudiée dans les chapitres consacrés à nos problématiques de recherche.

L'extraction de l'activité est automatique : le programme réalisant cette extraction itère sur les *commits* entre la version v_i et la fin de la période considérée (la taille de la période est définie dans les différentes études). Pour chaque *commit*, le script extrait les différences introduites par celui-ci et enregistre le nombre de lignes modifiées par l'auteur du commit dans chacun des modules, mesure également appelée *code churn* [Munson et Elbaum, 1998]. Pour extraire ce *churn*, nous utilisons l'outil `git diff` pour calculer les modifications apportées par chaque *commit*. Cet outil effectue une comparaison textuelle du code source, en ignorant les espaces, tabulations ou lignes vides.

Différents traitements sont appliqués lors de l'extraction, afin d'améliorer la précision des mesures ; les deux ayant le plus d'impact sur l'activité sont présentés ci-dessous.

Élimination des *commits* de fusion

Lorsque les modifications d'une branche sont intégrées à une autre, il arrive qu'un *commit* soit automatiquement créé afin d'enregistrer la fusion entre les deux branches. Prenons par exemple un commit de fusion c_f ayant pour but de fusionner deux versions v_a et v_b dont l'historique de développement a divergé depuis une version v_c, comme illustré figure 3.6. Lorsque l'outil `git diff` va calculer les modifications apportées par le commit c_f à la version v_a, toutes les modification apportées depuis l'ancêtre commun v_c jusqu'à la version v_b seront renvoyées. Le résultat de ce comportement est que le *churn* de l'auteur du commit de fusion sera très élevé alors que dans la plupart des cas cette fusion est automatique, ce qui va donc fausser grandement les mesures d'activité extraites. Par exemple, le *commit* de fusion c_f de la figure 3.6 a un *churn* de 192 lignes (si l'on ignore les cas où des conflits ont lieu lors de la fusion). Les *commits* de fusion sont donc ignorés lors de l'extraction de l'activité des développeurs.

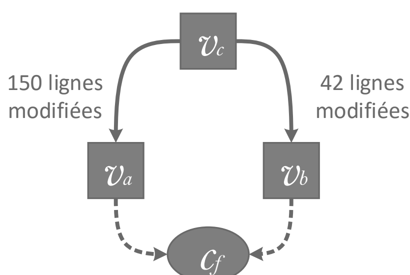

FIGURE 3.6 : Exemple de *commit* de fusion.

LISTING 3.1 : Exemple d'une trace de *commits* incluant une opération de renommage de fichiers. Les deux nombres avant le nom du fichier représentent le nombre de lignes de code ajoutées et respectivement supprimées par chaque *commit*.

```
1  commit 0
2  C 200 0 Test.rb
3  ---
4  commit 1
5  M 190 2 Test.rb
6  ---
7  commit 2
8  R 0 5 {Test => Hello}.rb
9  ---
10 commit 3
11 M 2 1 Hello.rb
12 ---
```

Détection des renommages de fichiers

Lorsqu'un fichier est renommé ou déplacé par un développeur, cette opération est enregistrée dans le *commit* correspondant comme la suppression d'un fichier avec l'ancien nom et l'ajout d'un fichier avec le nouveau nom. Si l'on prend par exemple la trace de *commits* du listing 3.1, le fichier *Test.rb* est renommé en *Hello.rb* dans le *commit* 2. La détection de ce renommage est optionnelle dans la configuration de Git et l'option correspondante doit être activée afin de la permettre. Supposons que nous voulons extraire le *churn* du fichier *Hello.rb* entre les *commits* 1 et 3 inclus :

– dans le cas où le renommage est détecté, le *churn* de *Hello.rb* est égal à la somme du nombre de lignes ajoutées ou modifiées dans ce fichier et dans le fichier `Test.rb` pour le *commit* antérieur au renommage, c'est-à-dire un total de 200 lignes ;
– dans le cas où le renommage n'est pas détecté, le *commit* 2 contient la création d'un fichier *Hello.rb* ayant un total de 383 lignes. Le *churn* du fichier *Hello.rb* sera donc

Tableau 3.1 : Extrait des données d'activité des développeurs produites par notre méthodologie.

Projet	Module	Développeur	*Churn*
Rails	`activesupport/lib/active_support/.*`	Bob	104
Rails	`activesupport/lib/active_support/cache/.*`	Bob	58
Rails	`actionpack/lib/action_controller/.*`	Alice	6
Rails	`actionpack/lib/action_controller/.*`	Charlie	3

de 386 lignes, en prenant en compte les trois lignes ajoutées ou supprimées du *commit* 3.

Il est donc nécessaire d'activer la détection de renommages lorsque l'historique des dépôts Git est parcouru.

Données produites

Cette étape de notre méthodologie produit donc un tableau de données contenant, pour chaque entrée :
- l'identité du module ;
- l'identité du développeur ;
- le niveau d'activité (*code churn*) du développeur sur le module.

Un exemple de ces données est présenté table 3.1.

3.3.6 Calcul des métriques d'organisation

Les métriques d'organisation des développeurs sont calculées en utilisant les données extraites ci-dessus. Ces métriques étant spécifiques à chacune des problématiques de notre thèse, elles seront présentées dans les chapitres liés à ces problématiques.

3.4 Analyse statistique des métriques

Nous présentons ici les principaux outils statistiques que nous utilisons pour nos contributions, à savoir le coefficient de corrélation de Spearman, le calcul d'intervalles de confiance avec la méthode *bootstrap* et la mesure de l'importance relative des métriques.

Les données en entrée de ces outils statistiques sont les valeurs de métriques, calculées à partir de l'activité des développeurs et le nombre de *commits* correctifs appliqués à chaque module. Nous introduisons ici les notations suivantes :
- μ, ν, ξ sont les différentes métriques décrivant l'organisation des développeurs. Nous utilisons seulement trois métriques pour décrire nos outils d'analyse statistique dans ce chapitre, mais nos contributions incluront un nombre plus large de métriques ;

- $\mu_0, \mu_1, ..., \mu_m$ sont les valeurs de la métrique μ pour les modules $0, 1, ..., m$;
- $\beta_0, \beta_1, ..., \beta_m$ sont les valeurs de la métrique de fiabilité β pour les modules $0, 1, ..., m$.

3.4.1 Corrélation entre métriques

La première étape de notre analyse statistique consiste à déterminer, pour chacune des métriques étudiées, si celle-ci est liée à la fiabilité logicielle. La force relation entre deux variables peut être mesurée via un coefficient de corrélation. Différentes formules existent pour calculer ce coefficient de corrélation, parmi lesquelles le coefficient de corrélation de Pearson [Pearson, 1895] et le coefficient de corrélation de Spearman [Spearman, 1904]. Pour cette mesure de la corrélation, nous utilisons le coefficient de Spearman, qui a l'avantage de ne pas être influencé pas les valeurs extrêmes observées et qui est plus adapté aux distributions de valeurs usuellement rencontrées en génie logiciel, qui sont fortement déséquilibrées [Louridas *et al.*, 2008].

Le coefficient de corrélation de Spearman se calcule à partir des rangs des valeurs des variables, et permet ainsi de mesurer une relation monotone et non-linéaire entre deux variables. Pour chaque projet, un ensemble M de modules a été défini et les valeurs des métriques μ et β sont calculées pour chaque module de l'ensemble. Chaque μ_i (respectivement β_i), est converti en un rang x_i (respectivement y_i), comme illustré table 3.2. Le coefficient de Spearman est ensuite calculé avec la formule suivante :

$$\rho = 1 - \frac{6\sum(x_i - y_i)^2}{|M|(|M|^2 - 1)}$$

La valeur de ρ est comprise dans l'intervalle $[-1;1]$ et s'interprète comme suit :

- plus ρ est proche de 0, moins les deux variables sont corrélées ;
- le signe de ρ indique le sens de la corrélation :
  - si la corrélation est positive, β augmente lorsque μ augmente,
  - si la corrélation est négative, β diminue lorsque μ augmente ;
- la taille d'effet de la corrélation peut être interprétée grâce à différentes échelles, dont celle de Cohen [Cohen *et al.*, 2002], illustrée table 3.3.

L'étude des coefficients de corrélation permet de réaliser une observation préliminaire de la relation entre les métriques d'activité et de fiabilité. Cependant, déduire des relations de cause à effet depuis les coefficients de corrélation obtenus n'est pas envisageable, étant donné que ceux-ci considèrent les métriques deux à deux, sans prendre en compte l'influence de variables tierces.

Intervalles de confiance avec la méthode *bootstrap*

Les coefficients de corrélation produits par la méthode ci-dessus étant basés sur un échantillon de modules logiciels, ils ne produisent que des estimations des corrélations. Il est donc nécessaire de déterminer la précision de ces estimations.

Tableau 3.2 : Calcul des rangs des valeurs de métriques pour la corrélation de Spearman. $\rho = 0,825$

μ_i	x_i	β_i	y_i	$x_i - y_i$
1	1	5	2	-1
5	2,5	6	3	-0.5
5	2,5	2	1	1.5
11	4	40	4	0
150	5	45	5	0

Tableau 3.3 : Table de Cohen pour l'interprétation du coefficient de corrélation.

$\lvert\rho\rvert$	Taille d'effet
≥ 0.10	Petite
≥ 0.30	Moyenne
≥ 0.50	Grande

Une statistique souvent utilisée afin de déterminer la confiance que l'on peut avoir dans les coefficients de corrélation est la valeur-p, qui correspond à la probabilité d'obtenir le même résultat dans le cas où l'hypothèse nulle du test statistique réalisé est vraie. Cette statistique est cependant difficile à interpréter et peut être remplacée par un intervalle de confiance. Un intervalle de confiance $[a;b]$ a une probabilité d'erreur α (ou une confiance $1-\alpha$). Nos résultats seront présentés avec deux intervalles de confiance ayant des niveaux de confiance de respectivement 90% et 95%. Prenons par exemple la figure 3.7, extraite de la figure 4.2 présentée page 55 : le coefficient de corrélation entre le nombre de lignes de code et le nombre de commits correctifs est représenté, pour chaque projet, par le point. Cette corrélation est d'environ 0,55 pour le projet Angular.JS. Les intervalles de confiance sont représentées par les barres : les barres les plus larges, situées à l'extérieur, correspondent à l'intervalle de confiance à 95% et les barres intérieures correspondent à l'intervalle de confiance à 90%. Dans le cas d'Angular.JS il y a une probabilité de 90% pour que la vraie valeur du coefficient de corrélation, estimée à partir des modules d'Angular.JS, soit entre 0,16 et 0,76 et il y a une probabilité de 95% pour que cette valeur soit entre 0,05 et 0,79.

Afin de calculer les intervalles de confiance pour les coefficients de corrélation de Spearman, nous utilisons la méthode *bootstrap* [Efron, 1979; Canty et Ripley, 2013]. Le processus utilisé par la méthode *bootstrap* est illustré figure 3.8 : depuis l'échantillon de N modules que nous avons tiré depuis l'univers des modules logiciels, *bootstrap* réalise B échantillons (notre implémentation utilise $B = 1000$) de N éléments, construits via un tirage aléatoire avec remise. La statistique estimée (par exemple le coefficient de corrélation ρ) est calculée pour chacun des B échantillons.

Une fois les B valeurs de la statistique calculées, il est possible de déterminer un intervalle de confiance en fonction de la distribution de ces valeurs, qui est donc une distributions de 1000 coefficients de corrélation. La statistique la plus simple consiste à utiliser des quantiles de la distribution afin de déterminer les limites de l'intervalle de confiance.

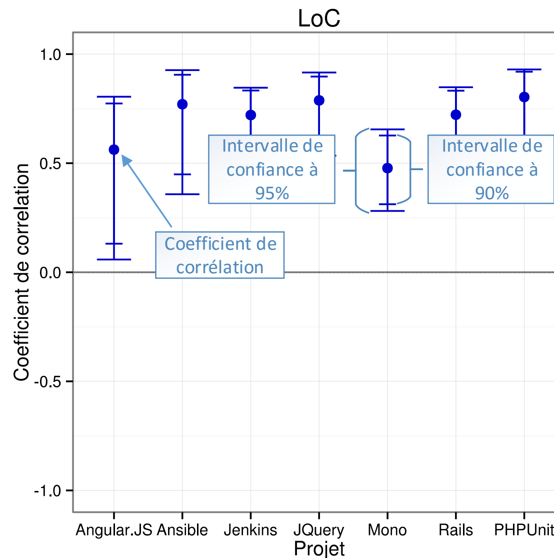

FIGURE 3.7 : Exemple des résultats de corrélation : corrélation entre le nombre de lignes de code d'un module et le nombre de *commits* correctifs de celui-ci.

Nous utilisons la statistique la plus avancée définie par Efron, appelée BCa[2] [Efron, 1987], qui permet d'ajuster automatiquement les résultats en fonction de la déformation de la distribution, tout en restant efficiente en termes de temps de calcul.

L'avantage de cette méthode, comparée à d'autres techniques statistiques, est qu'elle repose uniquement sur la liste de valeurs des métriques et qu'aucune supposition n'est faite sur la distribution de ces valeurs.

3.4.2 Régression multiple et importance relative des métriques

Afin de prendre en compte l'impact des différentes métriques d'activité sur la fiabilité logicielle, nous construisons des modèles de régression multiple. Une régression multiple consiste en un modèle ayant plusieurs variables d'entrée, appelées variables indépendantes ou *prédicteurs* et une variable de sortie appelée variable dépendante. Dans notre cas, les variables indépendantes sont les différentes métriques d'activité et la variable dépendante est la métrique de fiabilité choisie.

Un modèle de régression multiple se présente sous la forme $\beta = a\mu + b\nu + c\xi + d$ où a, b, c et d sont les paramètres à estimer en fonction des observations réalisées sur les modules d'un projet. La précision du modèle et de chacun des paramètres peut ensuite être estimée,

2. pour *bias-corrected and accelerated*

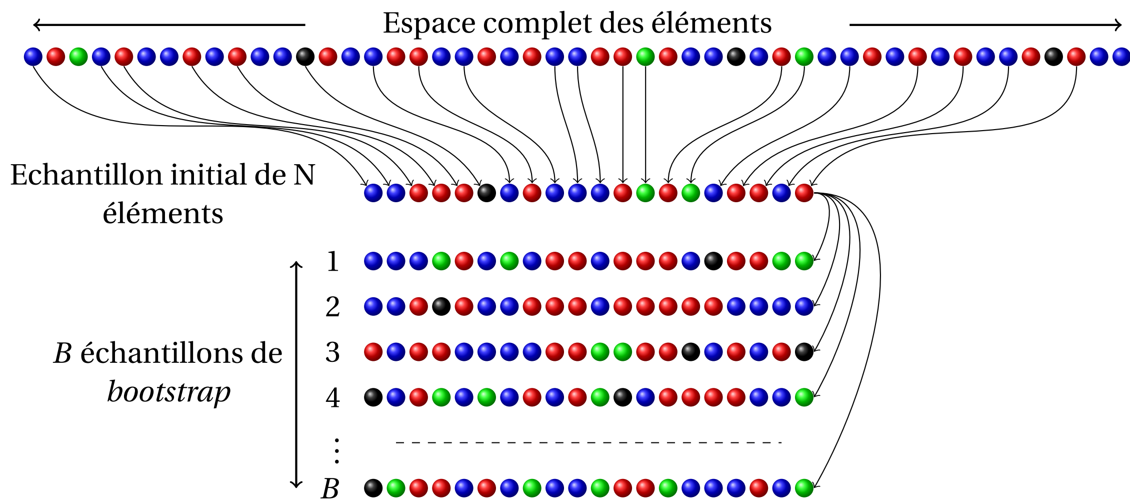

FIGURE 3.8 : Première étape de la méthode *bootstrap*

et, sous certaines conditions, il est possible d'évaluer l'influence individuelle des variables indépendantes.

La précision globale du modèle est évaluée via le coefficient de détermination ou R^2. Cependant, notre but est de mesurer l'importance de la contribution de chacune des variables indépendantes à la précision de ce modèle. Afin d'identifier les variables importantes pour la précision du modèle, plusieurs techniques consistent à ajouter ou supprimer des variables indépendantes du modèle, et à comparer les R^2 obtenus avec chacun des ensembles de variables. Cependant, l'ordre dans lequel les variables sont ajoutées ou supprimées n'est pas anodin dans le cas où les variables indépendantes ne sont pas orthogonales, ce qui est souvent le cas lorsqu'il s'agit de métriques logicielles. Nous utilisons donc la méthode appelée PMVD [3], qui consiste à réaliser une moyenne pondérée des gains de précision apportés par chaque variable, en prenant en compte tous les ordonnancements possibles [Groemping, 2006].

3.4.3 Synthèse

Notre procédé d'analyse statistique prend en entrée les valeurs des métriques d'organisation et de fiabilité obtenues pour les modules d'un projet donné. À partir de ces données, nous évaluons la relation entre chacune des métriques d'organisation et de fiabilité, via le calcul de coefficients de corrélation de Spearman et des intervalles de confiances calculés avec la méthode *bootstrap*. Nous construisons également un modèle multivarié ayant pour variable dépendante une métrique de fiabilité et évaluons l'importance de chacune des métriques d'activités dans ce modèle.

3. *Proportional marginal variance decomposition*

Il est important de noter que l'analyse que nous portons sur l'activité des développeurs et la fiabilité logicielle n'est pas seulement statistique, comme nous le verrons dans les chapitres suivants. Nos travaux contiennent également une part d'analyse qualitative, réalisée manuellement, qui permet de mieux comprendre les phénomènes que nous observons.

3.5 Réplication des résultats

Une part importante de la recherche empirique est la réplication des résultats, sans laquelle il est impossible de valider les théories exposées dans les études réalisées. En effet, les sujets de l'étude, les métriques et les outils statistiques utilisés sont autant de facteurs pouvant affecter les conclusions de nos travaux. Ces facteurs menacent ce que l'on appelle la validité externe de l'étude, c'est-à-dire la généralisation de ses résultats. C'est pourquoi il est nécessaire de tester les hypothèses émises dans nos travaux dans différentes conditions, afin de valider et d'améliorer les théories associées à celles-ci.

Dans le but de faciliter cette tâche, une de nos contributions est la mise à disposition du jeu de données utilisé lors de notre thèse, ainsi que de la suite logicielle utilisée pour l'analyse et l'extraction de ces données.

3.5.1 Données

Les données mises à disposition sont composées de toutes les données présentées dans le diagramme des figures 3.3 et 3.4, à savoir, pour chacun des projets de notre jeu de données :

- la liste des modules logiciels à la version v_i ;
- la liste des alias de développeurs devant être fusionnés ;
- la liste des *commits* correctifs présents dans la branche de maintenance de v_i ;
- la liste des contributions réalisées durant la phase de production de v_i ;
- les valeurs des métriques d'organisation et de fiabilité, pour chacun des modules ;
- les résultats de l'analyse statistique, qui seront également présentés dans les chapitres suivants.

3.5.2 Suite logicielle

Bien qu'une partie de notre procédé d'extraction soit réalisée manuellement, un certain nombre d'actions est exécuté automatiquement, pour lesquels nous avons développé une suite logicielle. Le processus d'extraction des métriques utilise un outil développé au sein de notre équipe de recherche, appelé DIGGIT. Cet outil, développé dans le langage Ruby, permet l'extraction d'information depuis un dépôt Git. Les méthodes développées afin d'extraire les métriques de fiabilité et l'activité des développeurs sont incluses dans

Tableau 3.4 : Projets sélectionnés dans notre jeu de données

Projet	Langage	v_i	Modules	Commits correctifs	Lignes de code
Angular.js	JavaScript	1.0.0	26	147	11 041
Ansible	Python	1.5.0	29	62	50 553
Jenkins	Java	1.509	60	74	79 774
JQuery	JavaScript	1.8.0	23	46	5 306
Mono	C#	2.10.0	184	351	1 777 719
PHPUnit	PHP	3.6.0	16	46	11 885
Rails	Ruby	2.3.2	46	390	33 919

un paquet `gem` [4]. Le calcul des métriques d'organisation et l'analyse statistique est contenu dans des scripts développés dans le langage R.

3.5.3 Projets sélectionnés

La recherche de projets *open-source* ayant un gestionnaire de versions et une architecture de branche de maintenance appropriés a été réalisée manuellement. Nous avons sélectionné sept projets répondant aux critères exposés dans la section 3.2 :

- AngularJS, un *framework* permettant la réalisation d'applications web, développé en JavaScript ;
- Ansible, un moteur d'automatisation de déploiement d'applications, développé en Python ;
- Jenkins, un serveur d'intégration continue, développé en Java ;
- JQuery, une bibliothèque permettant l'écriture de scripts clients et le parcours et modification du DOM [5], écrite en JavaScript ;
- Mono, une implémentation du langage C# et de la plateforme .NET, développée en C# ;
- Rails, un *framework* permettant la réalisation d'applications web suivant le modèle MVC [6], écrite en Ruby ;
- PHPUnit, un *framework* de tests unitaires, écrit en PHP.

Quelques informations concernant la taille de ces projets sont listées dans la table 3.4.

4. https://rubygems.org/gems/diggit-developers_activity
5. Document Object Model
6. Model-View-Controller

3.6 Validité de la méthodologie

Dans cette section nous parcourons les différents facteurs pouvant impacter la validité des résultats issus de cette méthodologie. Nous nous intéressons ici à la validité conceptuelle des résultats, qui décrit la représentativité des mesures générées par cette méthodologie vis-à-vis des phénomènes que nous souhaitons observer.

3.6.1 *Pull requests* et revue de code

Les projets que nous avons sélectionnés sont tous hébergés sur la plateforme GitHub. Sur cette plateforme, les contributions peuvent être apportées en utilisant le système de *pull requests* [Kalliamvakou *et al.*, 2014]. Lorsqu'un contributeur soumet une *pull request*, les *commits* qu'il souhaite voir appliqués sont soumis à la revue des autres développeurs du projet. Ces autres développeurs, par leurs commentaires, peuvent contribuer à l'amélioration du code source produit par le contributeur ayant créé la *pull request*. Ces contributions ne sont pas prises en compte par notre méthodologie. Pour pallier à cette limitation, il serait nécessaire de développer une méthode permettant d'évaluer les contributions apportées par les développeurs lors de commentaires au sein des *pull requests*.

3.6.2 Origine du code réusiné

Les modules listés pour chacun des projets correspondent à l'état du code source à la version v_i de chacun des projets. Or, il est possible que cette liste de modules ait évolué dans la période prise en compte pour calculer les mesures d'activité des développeurs. Dans le cas où ces modules sont affectés par des opérations de *refactoring* (ou « réusinage ») telles que le renommage ou le déplacement d'éléments de code source, il est possible que les contributions de certains développeurs soient perdues. Notre méthodologie prend en compte les opérations de renommage ou de déplacement de fichiers grâce à l'algorithme de *Git*, ce qui permet de retrouver l'origine des fichiers renommés ou déplacés. Cependant, les *refactorings* ayant lieu à un niveau de granularité plus fin ne sont pas pris en compte par notre procédé d'extraction.

3.6.3 Découpage du code en modules

Les métriques extraites par cette méthodologie sont à l'échelle des modules logiciels que nous avons déterminés en fonction de la hiérarchie de dossiers et de fichiers créée par les développeurs. Cette hiérarchie peut donc affecter les résultats et il est possible que l'activité extraite soit concentrée dans un module qui contient en fait différents concepts et pourrait être découpé plus finement. Une alternative à notre approche manuelle de découpage de modules serait l'utilisation d'une analyse sémantique du code source basée

sur les noms de variables ou de classes, permettant de regrouper les artefacts logiciels les plus proches [Kuhn *et al.*, 2007].

3.6.4 *Commits* correctifs et bogues

Nous avons choisi de nous passer des informations fournies par les systèmes de suivi de bogues des projets de notre corpus, du fait des incertitudes concernant la fiabilité des informations qu'ils fournissent [Bird *et al.*, 2009a; Herzig *et al.*, 2013]. Bien que ce procédé a des avantages, plusieurs incertitudes subsistent quant à sa validité :

- le nombre de *commits* correctifs peut ne pas correspondre au nombre de bogues corrigés : certains bogues peuvent requérir plusieurs *commits* afin d'être corrigés ;
- notre procédé ne prend en compte que les bogues corrigés pendant le temps où les versions v_i des projets étaient maintenues. Bien que les branches de maintenance observées sont des branches de support à long terme, il n'est pas impossible que des bogues soient toujours présents à la fin de ce support ;
- la date d'apparition des bogues dans le projet n'est pas considérée par notre procédé, ce qui peut amener à considérer des métriques d'organisation basées sur une période ultérieure à l'apparition de certains bogues ;
- la présence des *commits* correctifs sur la branche de maintenance repose sur la rigueur des développeurs : il est possible que des bogues corrigés pendant la production d'une version ultérieure à v_i n'aient pas tous été transférés sur la branche de maintenance, bien que nous ayons observé ces transferts pour tous les projets de notre corpus.

CHAPITRE

4

Répartition des contributions

Différentes métriques ont été définies avec pour objectif d'évaluer la répartition des contributions des développeurs. Les métriques de propriété du code permettent de distinguer les contributeurs majeurs et mineurs d'un module logiciel. La relation entre ces métriques et la fiabilité logicielle a été établie mais seulement dans un contexte industriel. La métrique de concentration de l'activité du module évalue le niveau d'interaction entre les développeurs et les modules logiciels. Cette métrique a été évaluée dans un contexte open-source*. Notre objectif dans ce chapitre est d'évaluer la relation entre répartition des contributions et fiabilité logicielle dans le cadre de projets* open-source*, tout en apportant une comparaison entre ces métriques et d'autres métriques logicielles définies précédemment. Pour ce faire, nous utilisons la méthodologie définie dans le chapitre précédent. Bien que nos résultats confirment la relation entre certaines des métriques étudiées et la fiabilité logicielle, il apparaît que ces métriques ont une importance faible lorsqu'on les compare à des métriques plus « classiques », telles que le nombre de lignes de code, le nombre de développeurs, ou la quantité de modifications appliquées à un module.*

Sommaire

4.1 Introduction 48
4.2 Définitions des métriques 49
4.3 Théories 52
4.4 Méthodologie 54
4.5 Analyse des résultats 54
4.6 Validité de l'étude 59
4.7 Limites et travaux futurs 61

4.1 Introduction

Différentes études ont montré que les métriques basées sur l'activité des développeurs ont une relation forte avec la fiabilité logicielle et qu'elles sont plus utiles que les métriques se basant sur le design du code lorsqu'il s'agit de prédire la quantité de bogues [Rahman et Devanbu, 2013]. Parmi les métriques définies précédemment, nous nous intéressons aux métriques de propriété du code définies par Bird *et al.* [Bird *et al.*, 2011] et à la métrique de concentration de l'activité définie par Posnett *et al.* [Posnett *et al.*, 2013].

Les métriques de propriété du code, appelée *code ownership* en anglais, se basent sur le niveau de contribution de chacun des développeurs sur les modules d'un projet. Les développeurs sont alors séparés en développeurs *majeurs* et *mineurs*, en fonction de ce niveau de contribution [Bird *et al.*, 2011].

La métrique de concentration de l'activité, appelée *module activity focus* en anglais, s'inspire de mesures du domaine de l'écologie ayant pour but d'évaluer le niveau d'interaction entre différentes espèces. Dans notre cas, cette métrique évalue le niveau de contribution des développeurs sur les modules d'un logiciel [Posnett *et al.*, 2013].

L'étude des mesures de propriété du code a été réalisée sur deux systèmes d'exploitation de Microsoft qui sont Windows Vista et Windows 7 [Bird *et al.*, 2011]. Bird *et al.* ont montré que les métriques de propriété du code ont une relation forte avec le nombre de bogues d'un module logiciel et qu'ajouter ces métriques à un modèle multivarié ayant le nombre de bogues en variable dépendante augmente sa précision. Plus le nombre de développeurs mineurs contribuant à un module est élevé, plus ce dernier contiendra de bogues. Une explication possible de ce phénomène vient du fait que les développeurs mineurs ont une connaissance moindre des modules auxquels ils contribuent et qu'ils introduiront donc plus de bogues. De plus, Bird *et al.* ont observé que deux autres métriques sont liées au nombre de bogues : le nombre de développeurs majeurs et le ratio de contributions ajoutées par le développeur principal du module vis à vis de la quantité totale de contributions sur ce module. De hautes valeurs de ces deux métriques caractérisent une forte propriété du code et sont liées à un nombre de bogues moins important. Contrairement aux développeurs mineurs, les développeurs majeurs ont plus de connaissances sur les modules auxquels ils contribuent et peuvent introduire moins de bogues.

Ces résultats ont deux conséquences principales :

- les équipes de développement devraient être réorganisées dans le but d'augmenter la propriété du code en limitant le nombre de développeurs mineurs ou, si ce n'est pas possible, en faisant en sorte que le code des développeurs mineurs soit relu par les développeurs majeurs ;
- les métriques de propriété du code devraient être utilisées afin de prédire le nombre de bogues dans les modules logiciels, étant donné que leur introduction dans des modèles augmente leur précision de manière significative.

L'étude de la mesure de concentration de l'activité a été réalisée sur un ensemble de projets de la fondation Apache [Posnett *et al.*, 2013]. Cette étude a montré dans un premier

temps que plus l'activité sur un module est concentrée, moins celui-ci aura de bogues. Cependant, lorsque cette métrique est utilisée dans un modèle multivarié prenant en compte des métriques telles que le nombre de développeurs et la taille des fichiers, il apparait que l'effet de la concentration de l'activité est inversé, c'est-à-dire que plus un fichier reçoit des contributions de développeurs concentrés, plus il aura de bogues. Les auteurs de l'étude attribuent ce résultat au fait que les fichiers sur lesquels les développeurs sont les plus concentrés peuvent être plus complexes, ce qui entraine un nombre plus élevé de bogues.

L'objectif de notre contribution dans ce chapitre est de valider l'intérêt des différentes métriques dans un contexte différent des études originales, afin de pallier aux limitations de ces études relatives à leur généralisation. Dans ce chapitre, nous reproduisons ces études sur les projets *open-source* de notre jeu de données en utilisant la méthodologie définie dans le chapitre précédent. Notre méthodologie permet, premièrement, de reproduire les résultats liés à la corrélation entre les métriques de répartition des contributions et la fiabilité. Ensuite, nous étendons les travaux précédents en réalisant un comparatif de l'importance des différentes métriques, permettant d'évaluer leur intérêt.

Ce chapitre est organisé comme suit : nous présentons les métriques permettant de quantifier la propriété du code (§4.2) puis les théories relatives à l'impact de cette propriété sur la fiabilité logicielle (§4.3). Nous présentons quelques éléments de méthodologie spécifiques à cette étude (§4.4). Nous présentons ensuite les résultats de notre étude, relatifs à la relation entre les métriques de propriété et la fiabilité logicielle (§4.5) et les points menaçant la validité de cette étude (§4.6). Ce chapitre se termine par une discussion concernant les limites et perspectives de cette étude (§4.7).

4.2 Définitions des métriques

Nous présentons ici les formules permettant de calculer les métriques utilisées dans ce chapitre.

4.2.1 Activité des développeurs

Les métriques de propriété du code ont pour base l'activité des développeurs sur les modules des projets de notre corpus. Soit D l'ensemble des développeurs contribuant à un projet et M l'ensemble des modules de ce projet. Nous définissons $A_{m,d}$ comme étant le niveau d'activité du développeur d sur le module m.

Afin de mesurer ce niveau d'activité, nous avons choisi de nous appuyer sur les informations fournies par le gestionnaire de versions, à savoir la quantité de lignes ajoutées, modifiées ou supprimées dans chacun des *commits* (cette mesure est connut sous le nom de *code churn* [Munson et Elbaum, 1998]). La définition du niveau d'activité d'un dévelop-

peur d sur un module m est donc la suivante :

$$A_{m,d} = \sum_{c \in C} (add(c,m) + del(c,m) + mod(c,m))$$

Dans cette définition, c représente un *commit*, C l'ensemble des *commits* considérés, et $add(c,m)$, $del(c,m)$, $mod(c,m)$ représentent respectivement le nombre de lignes ajoutées, supprimées ou modifiées dans le module m par le *commit* c.

4.2.2 Propriété du code

La propriété du code est définie comme le ratio d'activité d'un développeur sur un module comparé au reste des contributeurs. De manière plus formelle, la propriété du développeur (*ownership* en anglais) d sur le module m est :

$$own_{m,d} = \frac{A_{m,d}}{\sum_{d' \in D} (A_{m,d'})}$$

De cette mesure de la propriété du code découlent trois métriques définies par Bird *et al.* [Bird *et al.*, 2011]. Nous illustrons les définitions de ces trois métriques grâce à la figure 4.1, qui donne un exemple de la répartition des contributions sur deux modules fictifs.

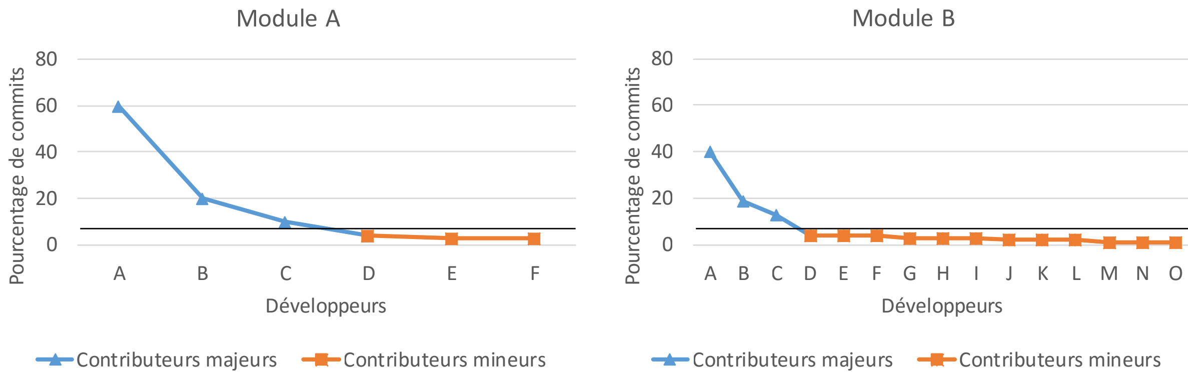

FIGURE 4.1 : Répartition des contributions dans deux modules logiciels fictifs.

Définition 4.1 (MVO) *La métrique MVO, pour* Most Valuable Owner, *correspond à la propriété maximale d'un module, c'est-à-dire la plus haute valeur de propriété obtenue par un développeur de ce module. De façon plus formelle, la propriété maximale d'un module m est :*

$$MVO_m = max(\{ own_{d,m} \mid d \in D \})$$

Dans la figure 4.1, cette mesure correspond au pourcentage de *commits* réalisé par le développeur le plus à gauche, soit $0,60$ pour le module A et $0,40$ pour le module B.

Définition 4.2 (Minor) *Les contributeurs mineurs d'un module sont ceux ayant réalisé 5% ou moins des contributions du module. La valeur de cette métrique pour un module m est le nombre de contributeurs mineurs du module :*

$$Minor_m = |\ \{0 < own_{d,m} \leq 0.05 \mid d \in D\}\ |$$

Dans la figure 4.1, cette mesure correspond au nombre de développeurs représentés par des carrés, ayant un pourcentage de contributions inférieur à 5%, soit 3 développeurs pour le module A, et 12 pour le module B.

Définition 4.3 (Major) *Les contributeurs majeurs d'un module sont ceux ayant réalisé plus de 5% des contributions du module. La valeur de cette métrique pour un module m est le nombre de contributeurs majeurs du module :*

$$Major_m = |\ \{own_{d,m} > 0.05 \mid d \in D\}\ |$$

Dans la figure 4.1, cette mesure correspond au nombre de développeurs représentés par des triangles, ayant un pourcentage de contributions supérieur à 5%, soit 3 développeurs dans chacun des modules.

Bird *et al.* ont fait varier le seuil de 5% qu'ils ont défini permettant de distinguer les contributeurs mineurs et majeurs. Avec des seuils variant entre 2% et 10%, ils ont obtenu des résultats sensiblement identiques vis-à-vis de la relation entre propriété du code et fiabilité logicielle. Nous conservons donc le seuil de 5% qu'ils ont utilisé.

4.2.3 Concentration de l'activité

La métrique de concentration de l'activité utilisée par Posnett *et al.* est égale au d' de Blüthgen *et al.*, qui est une mesure quantifiant l'interaction entre deux espèces animales ou végétales [Posnett *et al.*, 2013; Blüthgen *et al.*, 2006].

Cette mesure se base sur la matrice d'interaction entre les développeurs et les modules. Chaque ligne de cette matrice correspond à un développeur, chaque colonne à un module et la valeur d'une cellule correspond à l'activité d'un développeur sur le module. La valeur de la métrique de concentration de l'activité sur un module, ou MAF (pour *Module Activity Focus*), est comprise entre 0 et 1. Pour un module donné, cette métrique vaut 0 lorsque les développeurs contribuant à ce module sont dispersés, c'est-à-dire qu'ils contribuent à tous les modules du projet. Cette métrique vaut 1 lorsque le module reçoit des contributions de développeurs concentrés uniquement sur ce module.

La métrique MAF est calculée grâce aux formules suivantes :

$$MAF_m = \frac{\delta_m - \delta_{m_{min}}}{\delta_{m_{max}} - \delta_{m_{min}}}, \quad \delta_m = \left(- \sum_{d \in D} \frac{A_{m,d}}{A_m} ln \frac{A_d}{A} \right) - \left(- \sum_{d \in D} \frac{A_{m,d}}{A_m} ln \frac{A_{m,d}}{A_m} \right).$$

Dans ces formules, A_m représente le niveau d'activité sur un module, A_d le niveau d'activité du développeur d sur le projet, et A le niveau d'activité total sur le projet.

Cette métrique est équivalente au d' de Blüthgen *et al.*, qui est implémenté dans le paquet R `bipartite` [Dormann, 2011]. C'est cette implémentation qui est utilisée dans notre étude.

4.3 Théories

Lorsque le nombre de développeurs participant à l'élaboration d'un logiciel augmente, la charge de travail doit être répartie entre ces contributeurs. Il s'ensuit un débat concernant la propriété du code : doit-elle être forte ou partagée entre les développeurs ? Deux théories diamétralement opposées s'affrontent sur ce point :

- la première théorie, supportée par le mouvement de l'*eXtreme Programming* [Beck, 1999] et des figures de premier plan du développement *open-source* telles qu'Éric Raymond [Raymond, 1999], est en faveur d'une propriété du code partagée. Ce dernier a introduit la « Loi de Linus » (faisant référence au procédé de développement de Linux et à Linus Torvalds, son créateur) qui dit que « avec suffisamment d'yeux, tous les bogues sont superficiels », c'est-à-dire que l'augmentation du nombre de contributeurs accélérera la détection et la correction de bogues ;
- la seconde théorie est en faveur d'une propriété du code forte, en partant du principe qu'avec l'augmentation du nombre de contributeurs, la coordination des efforts de développement augmente fortement et qu'il devient nécessaire que certains développeurs « possèdent » certains modules logiciels et soient responsables des contributions réalisées sur ceux-ci.

Comme nous l'avons vu dans notre état de l'art, il est possible de trouver des résultats en faveur de ces deux théories, en fonction des métriques utilisées et des projets sur lesquels les études ont été réalisées. Les hypothèses formulées dans ce chapitre se basent sur la seconde théorie, étant donnée qu'elle est appuyée par les résultats de Bird *et al* et Posnett *et al*, dont nous utilisons les métriques.

Le jeu de métriques défini par Bird *et al* [Bird *et al.*, 2011] a pour objectif de distinguer les situations où la propriété du code est partagée entre de nombreux développeurs de celles où la propriété du code est forte.

Nous présentons ici les hypothèses découlant de leurs observations :

Most Valuable Owner. Cette métrique correspond à l'importance du développeur principal du module en termes de contributions. Si, pour un module m, MVO_m s'approche de

1, cela signifie que toutes les contributions sur le modules sont le fruit d'un seul individu. Lorsque MVO_m est bas cela signifie que le contributeur principal du module réalise une faible proportion des contributions et que la propriété du code du module est partagée entre les développeurs. Si l'on suit la théorie en faveur de la forte propriété du code, la relation entre la métrique MVO et la fiabilité logicielle est énoncée par l'hypothèse suivante :

Hypothèse 4.1
La métrique MVO est corrélée négativement avec le nombre et la densité de *commits* correctifs.

Minor. Dans le cas où un grand nombre de contributeurs mineurs est présent sur un module, un grand nombre de contributions au module est réalisé par des développeurs ayant peu d'expérience sur celui-ci et le module est partagé entre de nombreux développeurs. Le travail est fragmenté entre de nombreux développeurs et superviser les contributions de tous ces développeurs inexpérimentés peut s'avérer complexe. L'impact de la métrique $Minor$ sur la fiabilité logicielle est donc formulé comme suit :

Hypothèse 4.2
La métrique $Minor$ est corrélée positivement avec le nombre et la densité de *commits* correctifs.

Major. Lorsque le nombre de contributeurs majeurs est élevé, la coordination entre ceux-ci est plus difficile. Ces difficultés de synchronisation peuvent cependant être tempérées par le fait que les contributeurs majeurs ont une plus grande expertise du module. L'impact de la métrique $Major$ sur la fiabilité logicielle est donc formulé comme suit :

Hypothèse 4.3
La métrique $Major$ est corrélée positivement avec le nombre et la densité de *commits* correctifs et la taille d'effet de cette corrélation est relativement faible.

MAF. La métrique MAF décrit la concentration des développeurs sur un module. Une propriété forte du code allant de paire avec une forte concentration de l'activité, nous émettons l'hypothèse suivante concertant la relation entre MAF et la fiabilité.

Hypothèse 4.4
La métrique MAF est corrélée négativement avec avec le nombre et la densité de *commits* correctifs.

4.4 Méthodologie

Avant de présenter les résultats de notre analyse, nous abordons ici quelques points de méthodologie spécifiques à cette étude.

Métriques

Afin d'atteindre l'objectif de cette étude, qui est de statuer sur l'intérêt des métriques d'organisation des développeurs, nous devons calculer non seulement ces métriques mais aussi des métriques « de base » :

- le nombre de lignes de code du module (LoC) ;
- le *code churn* du module, c'est-à-dire la somme des lignes ajoutées, modifiées ou supprimées dans chacun des *commits* de la période considérée ;
- le nombre total de développeurs ayant modifié le module pendant la période considérée.

Ces métriques serviront d'éléments de comparaison pour évaluer l'importance des mesures d'organisation des développeurs.

Périodes de temps

Les travaux de Bird *et al* considèrent les modifications effectuées sur une période de temps correspondant au développement d'une version du logiciel depuis la version précédente. Afin de nous approcher au mieux de la méthodologie de l'étude que nous répliquons, nous analysons ici aussi l'activité des développeurs réalisée depuis la version précédente du projet. Les versions choisies sont listées dans la table 4.1.

4.5 Analyse des résultats

Cette section présente les résultats de l'analyse statistique que nous faisons de la relation entre les métriques de répartition des contributions et la fiabilité des modules logiciels. Nous commençons par évaluer la relation de chacune des métriques avec la fiabilité, puis nous étudions l'importance des métriques de répartition des contributions dans un modèle multivarié.

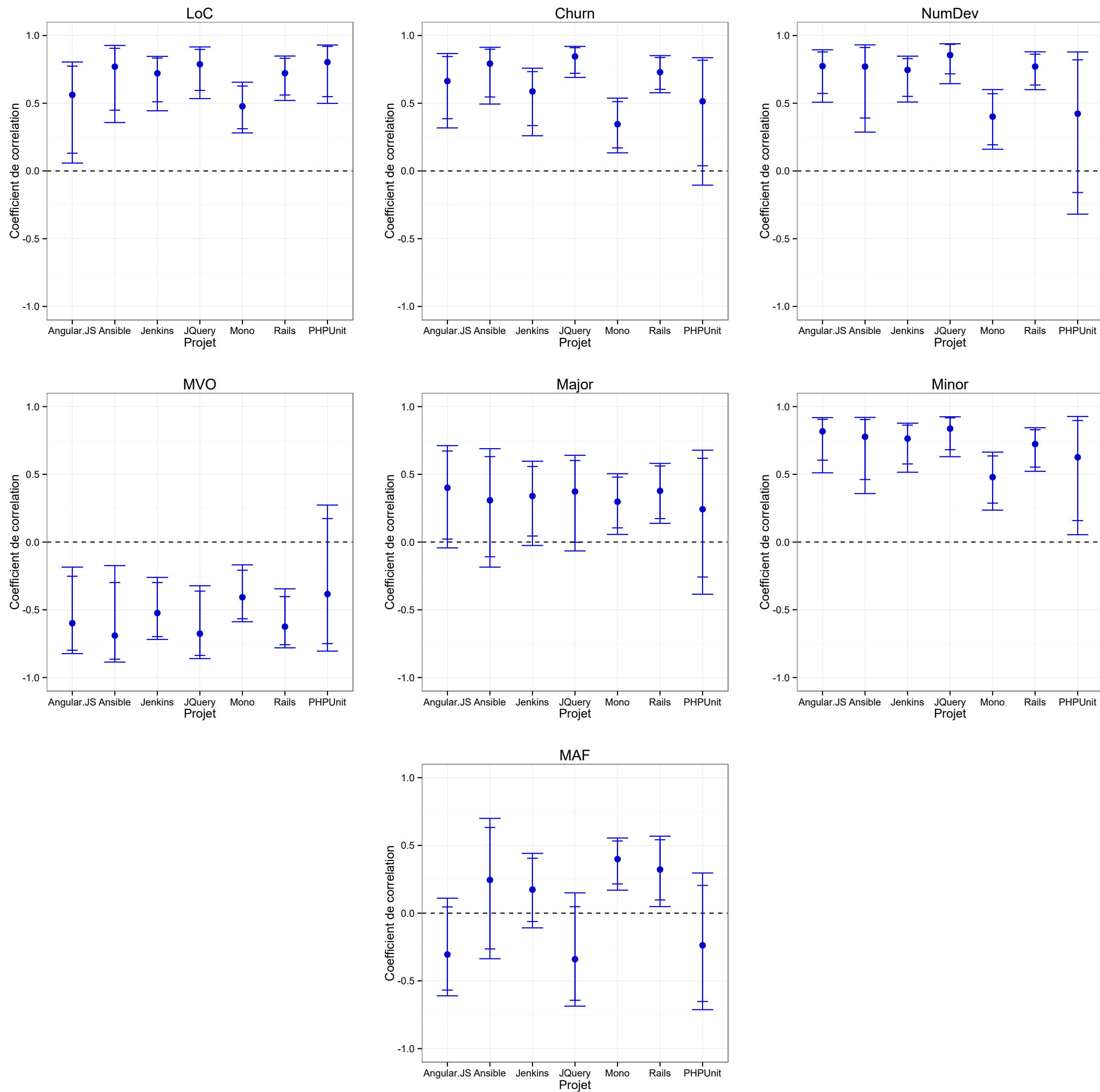

FIGURE 4.2 : Corrélation entre les métriques de répartition des contributions et **le nombre** de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*.

Tableau 4.1 : Versions délimitant les périodes considérées des projets.

Projet	Version v_0	Version v_{-1}
Angular.JS	1.0.0	0.10.0
Ansible	1.5.0	1.4.0
Jenkins	1.509	1.480
JQuery	1.8.0	1.7.0
Mono	2.10.0	2.8.0
PHPUnit	3.6.0	3.5.0
Rails	2.3.2	2.2.0

4.5.1 Relation entre les métriques de répartition des contributions et de fiabilité

Afin de déterminer si la relation entre les métriques de propriété du code et la fiabilité est présente dans notre corpus de projets, nous réalisons des tests de corrélation entre chacune des métriques décrites ci-dessus et la fiabilité des modules. Nous utilisons deux métriques pour la fiabilité : le nombre de *commits* correctifs, qui s'approche de la métrique utilisée par Bird *et al.* (le nombre de bogues corrigés) et la densité de *commits* correctifs, c'est-à-dire le nombre de *commits* correctifs par ligne de code du module.

Les figures 4.2 et 4.3 présentent les résultats obtenus avec la corrélation de Spearman, et les intervalles de confiance pour chaque coefficient de corrélation. Les tests de corrélation ont également été faits pour les métriques « de base » listées dans la section précédente afin de comparer les métriques de répartition des contributions à ces dernières.

Les résultats relatifs à la corrélation entre les mesures de propriété du code (la propriété maximale et les nombres de développeurs majeurs et mineurs) et le nombre de *commits* correctifs sont similaires à ceux décrits par Bird *et al.* [Bird *et al.*, 2011] :

- la métrique *MVO* est corrélée négativement avec le nombre de *commits* correctifs, c'est-à-dire que plus la propriété du code est forte, plus la fiabilité augmente ;
- la métrique *Major* est corrélée positivement avec la fiabilité, mais avec une taille d'effet relativement faible et des intervalles de confiance contenant des valeurs à la fois supérieurs et inférieurs à zéro dans plusieurs projets (rendant ces corrélations non significatives d'un point de vue statistique) ;
- la métrique *Minor* est corrélée positivement avec le nombre de *commits* correctifs, avec des corrélations relativement fortes.

Dans le cas de la métrique MAF, trois corrélations sont négatives alors que quatre sont positives. Les corrélations sont statistiquement significatives dans seulement deux cas sur sept et sont positives dans ces cas, c'est-à-dire que plus les développeurs sont concentrés, plus le nombre de bogues sera élevé dans ces deux projets (Mono et Rails).

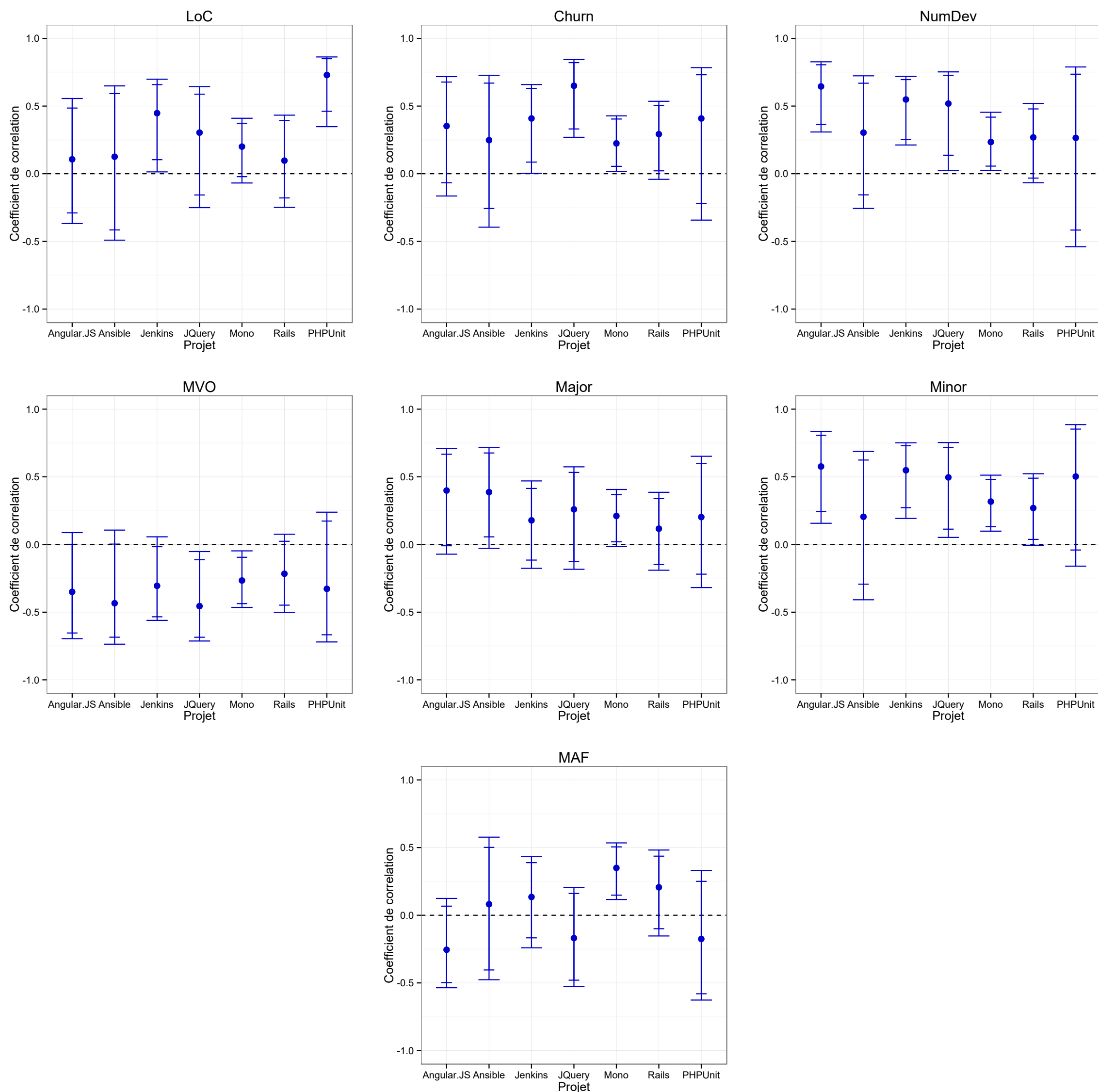

FIGURE 4.3 : Corrélation entre les métriques de répartition des contributions et **la densité** de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*.

Lorsque l'on prend en compte la densité de *commits* correctifs au lieu de leur nombre, la force de ces corrélations diminue fortement :

- la taille d'effet pour la propriété maximale et le nombre de contributeurs majeurs est moyenne ou faible. Les corrélations sont statistiquement significatives pour seulement deux projets dans le cas de la propriété maximale et un seul projet pour le nombre de contributeurs majeurs (Mono, si l'on considère l'intervalle de confiance à 90%) ;
- la taille d'effet diminue également pour le nombre de développeurs mineurs, mais reste relativement élevée dans certains cas. Les corrélations obtenues sont statistiquement significatives dans quatre projets (cinq si l'on considère un intervalle de confiance à 90%).

Dans le cas de MAF, les résultats sont similaires à ceux obtenus avec le nombre de *commits* correctifs.

Synthèse 4.5.1.1
En se basant sur ces résultats, nous pouvons conclure que les corrélations obtenues par Bird *et al.*, entre les métriques de propriété du code et le nombre de bogues, s'étendent aux projets *open-source*. Cependant, ces corrélations sont moins fortes et moins significatives d'un point de vue statistique lorsque l'on prend en compte la densité de *commits* correctifs. Les corrélations obtenues avec la métrique MAF varient grandement entre les projets, autant en termes de sens de la corrélation qu'en terme de signifiance statistique.

4.5.2 Importance relative des métriques

Bien que les métriques de répartition des contributions aient une relation avec le nombre de *commits* correctifs, cette relation est également présente pour d'autres métriques telles que le nombre de lignes de code, le *code churn* ou le nombre de développeurs d'un module. Notre but est de déterminer si les métriques de propriété du code ont une valeur ajoutée comparées à ces métriques de base et de déterminer si séparer les développeurs en contributeurs majeurs et mineurs est utile ou non.

Pour répondre à cette question, nous construisons un modèle de régression linéaire multiple par projet et regardons l'importance de chacune des métriques dans ces modèles. En utilisant un modèle de régression linéaire, il est possible de déterminer l'importance de chacune des métriques en regardant l'évolution du coefficient de détermination de la régression, noté R^2, qui correspond à la précision du modèle par rapport aux données. Pour déterminer l'importance de chacune des métriques, nous utilisons la méthode PMVD [Feldman, 2005], décrite dans le chapitre 3, en construisant un modèle avec les métriques « de base » et les métriques de répartition des contributions.

Avant d'appliquer la technique PMVD, nous avons observé que les métriques *nombre de développeurs* et *mineurs* sont colinéaires. La corrélation entre les valeurs de ces deux métriques, avec le coefficient de Pearson, varie de 0.896 à 0.996 selon le projet. Deux métriques avec une colinéarité aussi élevée peuvent être considérée comme identiques, et il est impossible d'utiliser les deux en même temps dans un modèle de régression linéaire. Cela signifie que, pour notre ensemble de projets, la métrique *Minor* est redondante avec le nombre de développeurs. Nous choisissons donc d'exclure *Minor* de la liste des métriques.

Il nous reste donc à déterminer l'importance relative des métriques *MVO*, *Major* et *MAF* vis-à-vis des métriques « de base », que sont le nombre de lignes de code, le *code churn* et le nombre de développeurs. Nous construisons donc un modèle de régression linéaire par projet contenant les métriques énoncées ci-dessus, avec le nombre de *commits* correctifs comme variable dépendante. La technique PMVD décompose le R^2 de chaque modèle en attribuant un score positif ou nul à chaque métrique. La figure 4.4 présente les résultats obtenus : chaque barre correspond à une métrique.

Dans chacun des sept projets, la métrique la plus importante est le nombre de développeurs ou le nombre de lignes de code du module. Les métriques MVO, le nombre de contributeurs majeurs et MAF ont un score très faible voire nul dans tous les projets.

Synthèse 4.5.2.2

En se basant sur ces observations, l'intérêt de calculer les métriques de répartition des contributions présentées dans ce chapitre dans les projets *open-source* est discutable. Le nombre de contributeurs mineurs est colinéaire avec le nombre de développeurs et les cas où MVO, le nombre de contributeurs majeurs ou MAF contribuent à la précision de modèles de régression linéaire multiple sont anecdotiques.

4.6 Validité de l'étude

Dans cette section nous discutons les différents facteurs pouvant affecter les résultats de notre étude. La validité conceptuelle ayant été abordée dans le chapitre 3, nous nous intéressons ici à la validité interne de l'étude qui étudie les biais pouvant affecter les résultats, c'est-à-dire les variables additionnelles pouvant expliquer nos résultats ou les différences vis-à-vis des études originales.

4.6.1 Contributeurs majeurs et mineurs

Dans les projets Windows étudiés par Bird *et al.*, la plupart des développeurs étaient des contributeurs majeurs d'au moins un module et seuls quelques individus étaient exclusivement des contributeurs mineurs [Bird *et al.*, 2011]. Les projets *open-source* suivent un modèle différent, avec une partie des développeurs qui sont des contributeurs principaux

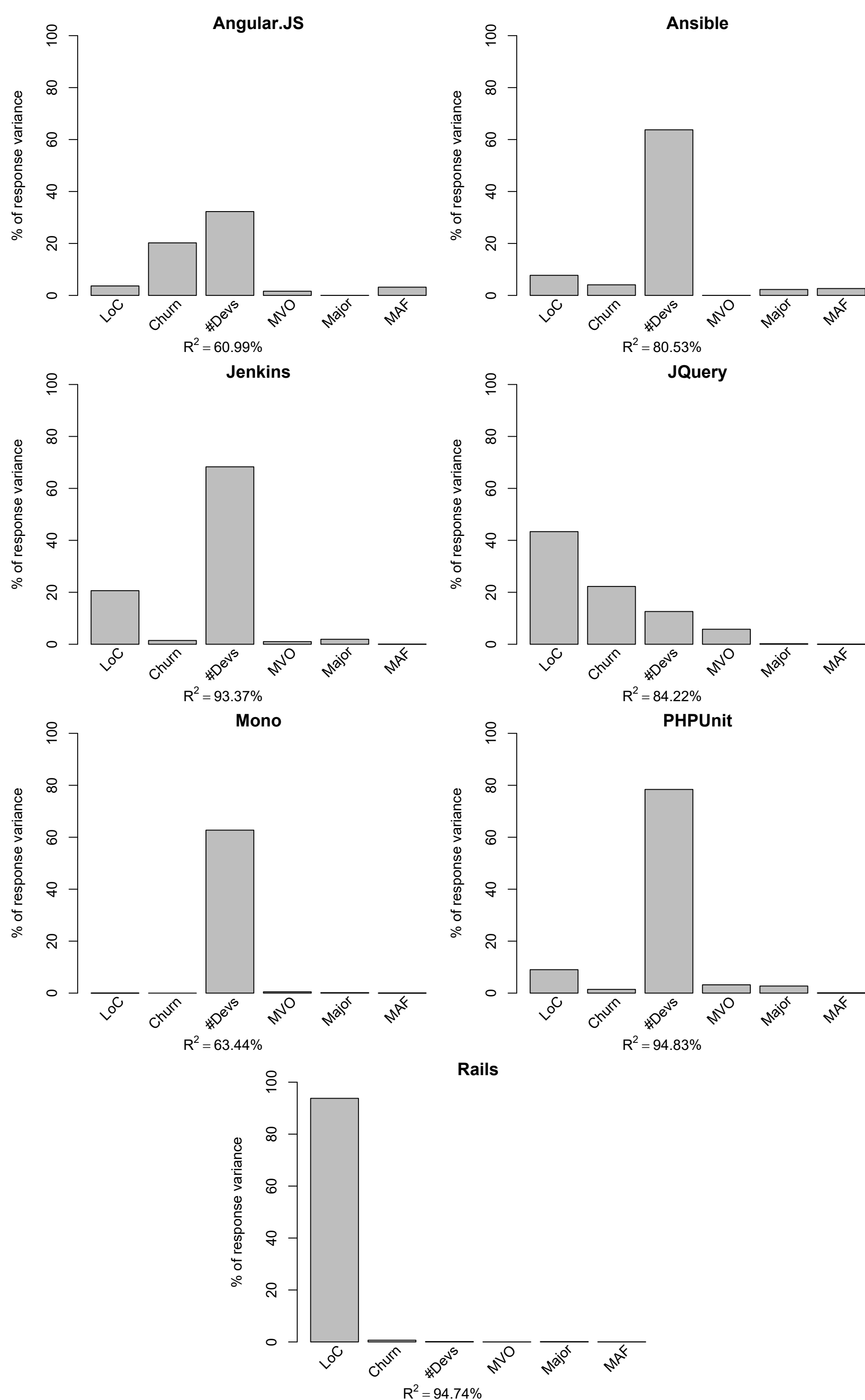

FIGURE 4.4 : Importance relative des métriques dans un modèle multivarié avec pour variable dépendante le nombre de *commits* correctifs.

du projet et une autre partie qui sont des contributeurs secondaires qui n'apportent qu'un petit nombre de contributions. Dans les projets de notre jeu de données, la proportion des développeurs n'étant que des contributeurs mineurs varie entre 50% et 79%. Les métriques définies dans cette étude ne faisant pas la différence entre les développeurs principaux et secondaires d'un projet, nous avons tenté de prendre en compte cette différence en séparant les contributeurs mineurs des modules en deux catégories : les contributeurs mineurs qui sont des contributeurs majeurs d'un autre module et les développeurs qui sont exclusivement des contributeurs mineurs. Les corrélations obtenues avec ces deux métriques ont une taille d'effet ou une signifiance statistique plus faible que les métriques de propriété du code existantes.

4.6.2 Lignes de conduite de la répartition des contributions

Chaque projet définit ses propres lignes de conduite vis-à-vis de la répartition des contributions. Les règles définies incluent le procédé de soumission des contributions, le style de code, etc. Comme l'ont indiqué Mockus *et al.*, les lignes de conduite peuvent également contenir des indications relatives à la répartition des contributions et à la propriété du code [Mockus *et al.*, 2002]. Dans certains cas, les propriétaires de chaque module sont définis par l'équipe centrale du projet et ont plus de responsabilités que les autres développeurs, telles que relire les soumissions de *patchs*, remplir des rapports de bogues, etc.

Nous avons recherché la présence de telles lignes de conduite dans notre corpus de projets. Dans la plupart des projets, il n'y avait aucune mention de la répartition des contributions ou de la propriété du code dans les lignes de conduite. Dans Jenkins, les lignes de conduite préconisent une propriété du code partagée et les contributeurs principaux ne se voient pas assigner de tâches particulières : le document relatif à la gestion de Jenkins indique que « les contributeurs centraux utilisent leur propre jugement pour déterminer ce sur quoi ils travaillent ». Dans Mono, les lignes de conduites sont en faveur d'une forte propriété du code et dictent que l'auteur d'une portion de code source devient automatiquement son propriétaire et que toute modification ultérieure de cette portion de code doit être discutée avec ledit propriétaire. Cependant, nous n'avons pas trouvé de ligne de conduite imposant une forte propriété comme décrit par Mockus *et al.* [Mockus *et al.*, 2002].

4.7 Limites et travaux futurs

Les résultats présentés dans cette étude arrivent à des conclusions différentes que les études de celles des études que nous avons reproduites. Les résultats concernant la colinéarité entre le nombre de contributeurs mineurs et le nombre total de développeurs sont fortement liés au contexte *open-source*, où de nombreux développeurs sont des contributeurs mineurs du projet. Il convient donc de noter que nos résultats ne contredisent

pas nécessairement ceux de Bird *et al.* et que leur application est cantonnée aux projets *open-source*. De plus, nous n'avons étudié la dichotomie entre les contributeurs majeurs et mineurs que via les métriques correspondantes. Nous n'avons pas encore étudié l'application de ces deux catégories de développeurs aux métriques de réseaux sociaux, comme l'ont fait Bird *et al.* dans leur étude [Bird *et al.*, 2011]. Les métriques MVO et le nombre de développeurs majeurs apportent de faibles contributions à la précision des modèles de régression dans deux des projets de notre corpus. Afin de confirmer que ces métriques peuvent être utiles dans un contexte de prédiction de bogues, nous voudrions étendre cette étude à un ensemble plus grand de projets. L'augmentation du nombre de projets permettrait également de comprendre les conditions nécessaires pour que les métriques *MVO* et *MAF* améliorent la précision des modèles de régression linéaire.

CHAPITRE

5 Turnover

La rotation des effectifs, ou turnover*, est un phénomène qui se matérialise par les arrivées et départs de membres d'une équipe ou des changements de rôle de ceux-ci. Bien que ce phénomène ait été étudié dans de nombreux environnements, le* turnover *des développeurs dans les projets* open-source *n'a été la cible que de quelques études. Dans ce chapitre, nous étudions les projets de notre corpus avec l'objectif d'explorer les différents schémas de* turnover *présents et de caractériser la relation de ce* turnover *avec la fiabilité logicielle. Nous définissons les notions de* turnover *externe et interne, qui correspondent respectivement aux mouvements des développeurs inter-projets et intra-projets. Après avoir défini les métriques permettant de quantifier le* turnover *dans les projets* open-source*, nous réalisons une analyse descriptive de ces phénomènes, ainsi qu'une évaluation de la relation entre les métriques de* turnover *et de fiabilité logicielle.*

Sommaire

5.1 Introduction 64
5.2 Impact théorique de la rotation des effectifs 65
5.3 Définition des métriques 66
5.4 Problématiques de recherche 70
5.5 Sélection des périodes de temps 70
5.6 Résultats 72
5.7 Validité de l'étude 83
5.8 Limites et travaux futurs 83

5.1 Introduction

Au fil de l'évolution d'un projet, l'équipe qui y contribue évolue, avec des collaborateurs qui arrivent, partent ou changent leur rôle dans le projet. Ce phénomène est appelé *turnover*. Le *turnover* a été étudié en sociologie des organisations et dans le domaine de la collaboration assistée par ordinateur, ce qui a amené le développement de plusieurs théories relatives à son impact. La théorie la plus communément admise est que le *turnover* a un impact négatif sur la performance et la qualité du travail fourni à cause de la perte d'expérience engendrée par le départ de membres d'une équipe et le fait que les nouveaux arrivants nécessitent une période d'apprentissage ou d'adaptation à l'équipe [Huselid, 1995].

Dans le contexte du développement logiciel, le *turnover* a été analysé par Mockus dans un projet industriel [Mockus, 2010]. Ses résultats indiquent que les départs de développeurs d'un projet ont un effet négatif sur la fiabilité de celui-ci mais que les nouveaux arrivant n'ont pas d'impact sur sa fiabilité. Le travail présenté dans ce chapitre a pour objectif d'évaluer la généralisation de ces conclusions aux projets *open-source*.

Pour cela, nous introduisons des métriques basées sur l'activité des développeurs qui permettent la mesure du *turnover* interne et externe dans des projets *open-source*. En séparant le code source d'un projet logiciel en différents modules, il est ainsi possible de mesurer non seulement les arrivées et départs de développeurs du projet (c'est-à-dire le *turnover* externe) mais aussi les mouvements de développeurs à l'intérieur du projet (c'est-à-dire le *turnover* interne).

En se basant sur les concepts définis dans ce chapitre, nous quantifions les niveaux de *turnover* rencontrés dans les projets *open-source*. Nos métriques se basent sur le niveau d'activité produit par les nouveaux arrivants ou les développeurs sortants, sans prendre en compte le nombre de développeurs de chaque catégorie, étant donné les grandes disparités de niveau de contributions rencontrées dans les projets *open-source*. Nous produisons ensuite une analyse empirique de la relation entre le *turnover* et la fiabilité logicielle et nous évaluons l'importance des métriques de *turnover* dans un modèle multivarié.

Ce chapitre est organisé comme suit : nous rappelons les théories liées au *turnover*, qui découlent de notre état de l'art, dans la section 5.2 ; nous définissons les acteurs du *turnover* et les métriques associées dans la section 5.3 ; nous formulons les problématiques de recherche abordées dans ce chapitre dans la section 5.4 ; nous présentons un point de méthodologie spécifique à ces métriques, à savoir la sélection des périodes de temps appropriées dans la section 5.5 et nos résultats dans la section 5.6 ; enfin, nous discutons la validité de l'étude dans la section 5.7 et les limites et perspectives de cette étude dans la section 5.8.

5.2 Impact théorique de la rotation des effectifs

Lors de notre état de l'art (chapitre 2), nous avons abordé de nombreuses études liées au *turnover* dans différents contextes. Bien que peu de ces études concernent le *turnover* dans les projets logiciels (et aucune n'étudie le lien entre *turnover* et fiabilité dans les projets *open-source*), nous avons observé que les conclusions sont similaires en sociologie des organisations et dans le domaine de la collaboration assistée par ordinateur. Il est donc juste de supposer que ces résultats peuvent s'étendre au génie logiciel et aux projets *open-source*. Nous formulons dans cette section les hypothèses que nous allons évaluer dans ce chapitre.

5.2.1 Arrivées de développeurs

Selon Argote et Epple, les nouveaux arrivants suivent une courbe d'apprentissage avant d'être efficaces dans leur travail [Argote et Epple, 1990]. Ils peuvent être encadrés lors de leur arrivée afin de faciliter cet apprentissage [Canfora *et al.*, 2012]. En se basant sur ces observations et les résultats de Rahman et Devanbu concernant la relation entre l'expérience des développeurs et la quantité de bogues qu'ils introduisent [Rahman et Devanbu, 2011], nous émettons l'hypothèse suivante :

Hypothèse 5.1
Une augmentation des arrivées de développeurs sur un module logiciel diminue sa fiabilité.

5.2.2 Départs de développeurs

Selon Argote et Epple, les départs de membres d'une équipe entrainent une perte de compétences et peuvent perturber l'environnement et la collaboration des membres restants [Argote et Epple, 1990; Dess et Shaw, 2001]. Dans le contexte du génie logiciel, cette perte de compétences a été liée à une diminution de la fiabilité logicielle [Mockus, 2010]. Notre hypothèse concernant les départs de développeurs est la suivante :

Hypothèse 5.2
Une augmentation des départs de développeurs sur un module logiciel diminue sa fiabilité.

5.2.3 Réorganisation interne du projet

Les mouvements de développeurs peuvent être internes à une même équipe et un même développeur peut contribuer à différents modules au fil de l'évolution du projet. Selon Thompson, ce *turnover* interne peut être bénéfique, car il permet à des individus d'exprimer de nouvelles compétences [Thompson, 1967]. Cet aspect de la rotation des effectifs n'a pas, à notre connaissance, été abordé en génie logiciel. Cependant, on peut supposer que plus un développeur peut exprimer de compétences différences dans un projet, plus sa connaissance de celui-ci est grande et il en va de même pour la qualité de ses contributions. Au vu des théories validées en sociologie des organisations, nous émettons l'hypothèse suivante :

Hypothèse 5.3
Une augmentation de la rotation interne des effectifs sur un module augmente sa fiabilité.

5.3 Définition des métriques

Afin d'étudier les différentes composantes du *turnover*, nous définissons dans cette section cinq métriques pouvant être extraites depuis l'historique de développement contenu dans un gestionnaire de versions.

5.3.1 Pré-requis

Étant donné le caractère bénévole de nombreux contributeurs aux projets *open-source*, il n'est pas possible d'utiliser d'informations contractuelles afin de déduire les arrivées et départs de développeurs, comme c'est le cas pour des projets industriels. La présence ou l'absence d'un développeur dans le projet peut cependant être déduite à partir des contributions qu'il ou elle ajoute dans le gestionnaire de versions.

Afin d'extraire les informations relatives au *turnover*, nous comparons les ensembles de contributeurs actifs sur les modules logiciels dans deux périodes de temps consécutives p_1 et p_2. Ces deux périodes sont donc liées à trois *commits* c_i, c_{i-1} et c_{i-2} tels que :

- p_1 commence avec le *commit* c_{i-2} et finit avec le *commit* c_{i-1} ;
- p_2 commence avec le *commit* c_{i-1} et finit avec le *commit* c_i ;
- le *commit* c_i correspond à la version v_i du projet, définie dans le chapitre 3. La version v_i est la version du logiciel ayant une branche de maintenance et pour laquelle nous avons extrait les *commits* correctifs.

La sélection des *commits* c_{i-1} et c_{i-2} dépend de la taille des périodes p_1 et p_2. Nous présentons dans la section 5.5 l'approche que nous utilisons afin de déterminer une taille appropriée pour ces périodes.

5.3.2 Acteurs du *turnover*

Dans cette section, nous définissons les ensembles de développeurs impliqués dans le *turnover* et les métriques associées à ces ensembles. Pour illustrer ces définitions, nous utilisons le schéma représenté figure 5.1. Deux niveaux de *turnover* sont considérés : le *turnover* externe et interne. Les développeurs impliqués dans ces deux types de *turnover* peuvent être des nouveaux arrivants ou des développeurs sortants. Enfin, nous définissons un dernier ensemble d'acteurs du *turnover* : les développeurs persistants qui contribuent de façon continue à un module logiciel.

Nous considérons comme nouvel arrivant, pour un module donné, un développeur ayant contribué à ce module pour la période p_2 sans y avoir contribué pendant la période p_1, qui la précède. À l'inverse, les développeurs sortants d'un module sont les développeurs y contribuant pendant la période p_1 mais pas pendant la période p_2. Ces choix concernant les périodes p_1 et p_2 correspondent à notre objectif, qui est de mesurer l'impact de la rotation des effectifs sur le code de la version v_i. En effet, un développeur arrivant dans la période p_2 peut influencer la fiabilité du code étant donné que ses contributions se situent juste avant la sortie de la version v_i. À l'inverse, un développeur sortant d'un module ou du projet pendant la période p_1 peut influencer la fiabilité du code par son absence et la perte de compétences de l'équipe pour le développement de la version v_i.

Nous utilisons ci-dessous la notation suivante : $D_{m,p}$ représente l'ensemble des développeurs ayant contribué à m pendant la période p, c'est-à-dire les développeurs qui sont auteurs d'au moins un *commit* affectant un fichier du module m. D_p représente l'ensemble des développeurs ayant contribué au projet dans la période p.

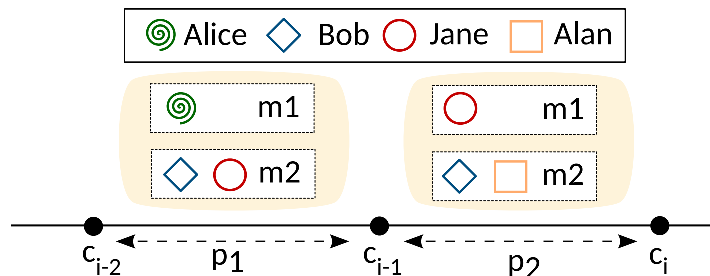

FIGURE 5.1 : Schéma illustrant la rotation des effectifs dans un projet fictif contenant deux modules.

Turnover externe

Le *turnover* externe correspond aux développeurs qui rejoignent ou quittent un projet. Les nouveaux arrivants externes d'un module m sont les développeurs ayant contribué au module entre c_{i-1} et c_i, mais qui n'ont contribué à aucun module du projet entre c_{i-2}

et c_{i-1}. L'ensemble des nouveaux arrivants externes est noté EN_{m,p_1,p_2} (pour « *external newcomers* »), et est calculé comme suit :

$$EN_{m,p_1,p_2} = D_{m,p_2} - D_{p_1}$$

Dans la figure 5.1, nous observons que Alan est un nouvel arrivant dans le module $m2$, et qu'il n'a contribué à aucun module pendant la période p_1. Il est donc un nouvel arrivant externe et EN_{m2,p_1,p_2} = {Alan}.

Les développeurs sortants externes d'un module m sont les développeurs ayant contribué au module pendant la période p_1, mais n'ayant apporté aucune contribution au code source pendant la période p_2. L'ensemble de développeurs sortants externes (ou « *external leavers* ») EL_{m,p_1,p_2} est calculé comme suit :

$$EL_{m,p_1,p_2} = D_{m,p_1} - D_{p_2}$$

Dans le schéma de la figure 5.1, nous observons que seule Alice a contribué au module $m1$ pendant la période p_1 mais qu'elle n'a pas contribué au projet pendant p_2. Par conséquent, EL_{m1,p_1,p_2} = {Alice}.

Turnover interne

Le *turnover* interne fait référence aux mouvements des développeurs à l'intérieur d'un projet. Bien que certains développeurs contribuent au projet pendant les deux périodes p_1 et p_2, il est possible qu'ils ne contribuent pas aux mêmes modules pendant les deux périodes.

Les nouveaux arrivants internes sont les développeurs ayant contribué à un module m pendant p_2 mais pas pendant p_1. Ces développeurs ont en revanche contribué à au moins un autre module du projet pendant p_1. Les nouveaux arrivants internes (ou « *internal newcomers* ») sont notés IN_{m,p_1,p_2} et cet ensemble est calculé comme suit :

$$IN_{m,p_1,p_2} = (D_{m,p_2} - D_{m,p_1}) \cap D_{p_1}$$

De la même manière que les exemples précédents, nous obtenons à partir du schéma de la figure 5.1 IN_{m1,p_1,p_2} = {Jane} et $IN_{m2,p_1,p_2} = \emptyset$.

Les développeurs sortants internes correspondent aux développeurs ayant cessé de contribué à un module mais qui sont toujours actifs dans le projet. Cet ensemble est noté IL_{m,P_1,P_2} (pour « *internal leavers* ») et est calculé comme suit :

$$IL_{m,p_1,p_2} = (D_{m,p_1} - D_{m,p_2}) \cap D_{p_2}$$

Nous observons figure 5.1 que Jane a modifié le module $m1$ pendant p_2 mais pas pendant p_1, tout en contribuant à $m1$ pendant p_1. Par conséquent, $IL_{m1,p_1,p_2} = \emptyset$ et IL_{m2,p_1,p_2} = {Jane}.

Développeurs persistants

Enfin, nous considérons les développeurs persistants qui sont les individus ayant contribué à un même module m pendant les deux périodes p_1 et p_2. Cet ensemble de développeurs, noté St_{m,p_1,p_2} (pour « *stayers* ») est défini comme suit :

$$St_{m,p_1,p_2} = D_{m,p_1} \cap D_{m,p_2}$$

Dans le schéma de la figure 5.1, seul Bob a contribué à un même module pendant les deux périodes p_1 et p_2. Nous avons donc $St_{m2,p_1,p_2} = \{ \text{Bob} \}$.

5.3.3 Définitions des métriques

L'objectif de nos métriques est de quantifier l'impact de chacune des catégories d'acteurs du *turnover* sur le code d'un module m. Étant donné les inégalités entre le niveau de contribution des différents développeurs d'un projet *open-source*, il n'est pas possible de quantifier l'importance d'une catégorie de développeurs en comptant le nombre d'individus qu'elle contient. Une alternative permettant de comparer les différentes catégories d'acteurs du *turnover* est de mesurer le niveau d'activité de chacune de ces catégories de développeurs, c'est-à-dire la quantité de code qu'ils produisent. L'activité d'un développeur est mesurée ici en utilisant le *code churn* de celui-ci [Munson et Elbaum, 1998]. Nous définissons une métrique par catégorie d'acteurs de la rotation des effectifs et chacune de ces métriques est calculée en utilisant l'activité de l'ensemble des individus de la catégorie correspondante.

Nos cinq métriques sont l'activité des développeurs sortants internes et externes (ILA et ELA), l'activité des nouveaux arrivants internes et externes (INA et ENA) et l'activité des développeurs persistants (StA) :

$$ILA_{m,p_1,p_2} = \sum_{d \in IL_{m,p_1,p_2}} A_{m,d,P_1}$$
$$ELA_{m,p_1,p_2} = \sum_{d \in EL_{m,p_1,p_2}} A_{m,d,p_1}$$
$$INA_{m,p_1,p_2} = \sum_{d \in IN_{m,p_1,p_2}} A_{m,d,p_2}$$
$$ENA_{m,p_1,p_2} = \sum_{d \in EN_{m,p_1,p_2}} A_{m,d,p_2}$$
$$StA_{m,p_1,p_2} = \sum_{d \in St_{m,p_1,p_2}} avg(A_{m,d,p_1}, A_{m,d,p_2})$$

En ce qui concerne l'activité des développeurs persistants, nous réalisons une moyenne de leur activité sur les deux périodes p_1 et p_2, dans le but d'avoir des métriques d'activité sur la même échelle et de pouvoir comparer l'activité des développeurs persistants à celle des autres catégories de développeurs.

5.4 Problématiques de recherche

Lors de notre état de l'art, nous avons trouvé peu d'études explorant les tendances du *turnover* à l'intérieur des projets *open-source*. Notre premier objectif est donc de chercher de telles tendances, en commençant par une vue globale au niveau des projets de notre corpus, puis en se concentrant sur le *turnover* ayant lieu sur les modules logiciels grâce aux métriques définies ci-dessus. Plus formellement, nous cherchons à répondre aux deux problématiques de recherche suivantes :

PR1
En se basant sur les définitions des nouveaux arrivants et développeurs sortants externes, est-ce que le *turnover* est un phénomène important dans les projets *open-source* ?

PR2
En regardant les contributions à l'échelle des modules logiciels, quels sont les schémas de contributions des acteurs du *turnover* ?

Les réponses aux questions ci-dessus fournissent un aperçu du *turnover*, aussi bien à l'échelle d'un projet qu'à celle de ses modules.

Notre analyse se poursuit en explorant la relation entre le *turnover* et la fiabilité logicielle, mesurée grâce au procédé décrit dans le chapitre 3. Notre troisième problématique de recherche est donc la suivante :

PR3
Y a-t-il une relation entre le *turnover* et la fiabilité d'un module logiciel ?

Lors de l'étude de cette problématique de recherche, nous répondons aux hypothèses énoncées dans la section 5.2.

5.5 Sélection des périodes de temps

Avant de présenter les résultats de cette étude, nous abordons un point de la méthodologie qui correspond à la sélection d'une valeur appropriée pour les périodes de temps utilisées dans les définitions de nos métriques.

Le calcul des métriques de *turnover* pour la version v_0 d'un projet repose sur le choix de deux périodes de temps p_1 et p_2. La longueur des périodes p_1 et p_2 peut avoir un impact sur les ensembles d'acteurs du *turnover* :

- si les périodes p_1 et p_2 sont trop courtes, cela pourrait amener à considérer comme nouveaux arrivants ou développeurs sortants des individus qui auraient arrêté de contribuer à un module pour une courte période de temps avant de recommencer ;
- si les périodes p_1 et p_2 sont trop longues, cela pourraient amener à considérer des développeurs comme étant persistants, alors qu'ils n'ont pas contribué à un module pour une longue période de temps et devraient être classifiés comme des développeurs sortants. De plus, plus ces périodes de temps sont longues, plus il est probable que nous considérions des contributions obsolètes pour la version v_i du logiciel. Enfin, il est possible qu'un développeur quitte un projet pour une longue période et le rejoigne : si cette période est suffisamment longue il ne doit pas être considéré comme un développeur persistant.

Exigence 5.5.1

Les périodes de temps sélectionnées doivent minimiser les erreurs de classification des acteurs du *turnover* tout en restant relativement courtes.

Nous avons choisi de tester trois configurations différentes, où les deux périodes p_1 et p_2 ont toutes deux des tailles de un, trois et six mois. Pour évaluer les erreurs commises dans l'identification des nouveaux arrivants dues au fait que les périodes de temps considérées sont trop courtes, nous comparons les ensembles de nouveaux arrivants (internes et externes) obtenus pour chaque module avec deux méthodes :

- le premier ensemble est celui qui sera utilisé par nos métriques, avec $|p_1| = |p_2|$;
- le deuxième ensemble a pour but de déterminer quels nouveaux arrivants ont en fait contribué au projet ou au module dans une période précédant p_1 (et ne devraient donc pas être considérés comme nouveaux arrivants). Pour cela, nous utilisons une période p_1 allant jusqu'au début de l'historique Git, c'est-à-dire que c_{i-2} correspond au premier commit du gestionnaire de versions Git.

Nous avons donc deux ensembles de nouveaux arrivants internes et deux ensembles de nouveaux arrivants externes pour chaque module, qu'il est possible de comparer en utilisant le coefficient de Sorensen-Dice [Dice, 1945], qui pour deux ensembles A et B est calculé comme suit :

$$s = \frac{2|A \cap B|}{|A| + |B|}$$

Ce coefficient de similarité est donc compris entre zéro et un, zéro signifiant que les deux ensembles A et B sont disjoints et un signifiant qu'ils sont identiques.

Pour évaluer les erreurs commises dans l'identification des développeurs sortants, le procédé est similaire à celui décrit ci-dessus, à l'exception que, pour le deuxième ensemble de développeurs sortants, nous utilisons une période p_1 de un, trois ou, six mois (selon la configuration qui est évaluée) et une période p_2 allant jusqu'à la fin de l'historique, c'est-

à-dire que c_i est le commit le plus récent du gestionnaire de versions Git (en restant sur la même branche).

Nous avons donc pour chaque module un coefficient de similarité et il est donc possible de tracer les distributions de ces coefficients de similarité pour chaque catégorie d'acteurs du *turnover* et chaque projet. Des exemples des distributions de coefficients sont illustrés figure 5.2, avec les résultats obtenus avec le projet Rails. L'ensemble complet de ces distributions est visible en annexe A.

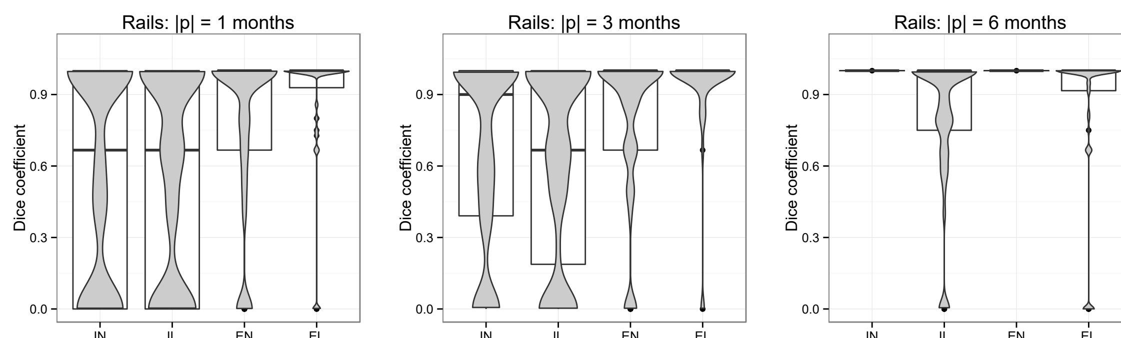

FIGURE 5.2 : Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique.

Cette figure montre par exemple qu'avec des périodes de un ou trois mois, un grand nombre de nouveaux arrivants ou développeurs sortants internes sont mal classifiés, comme l'indiquent les faibles coefficients de similarité observés.

Solution 5.5.1

Afin de choisir une période de temps appropriée, nous avons fixé un seuil pour la médiane de nos distributions qui est de 0.75. Nous avons donc sélectionné la période pour laquelle les médianes des distributions présentées en figure 5.2 et en annexe A sont toutes supérieures à ce seuil. Cette période est égale à six mois.

5.6 Résultats

Nous présentons dans cette section les résultats permettant de répondre à nos problématiques de recherche :

- nous évaluons tout d'abord le niveau de *turnover* rencontré dans les projets de notre jeu de données et l'évolution du nombre de nouveaux arrivants, développeurs sortants et persistants à l'échelle de ces projets ;

- ensuite, nous analysons la répartition des acteurs du *turnover* à l'échelle des modules logiciels ;
- enfin, nous mesurons la relation entre le *turnover* et la fiabilité logicielle, grâce aux métriques définies dans ce chapitre.

5.6.1 Rotation des effectifs à l'échelle du projet (PR1)

Afin de caractériser le *turnover* à l'échelle des projets, nous regardons l'évolution du nombre de nouveaux arrivants et développeurs sortants externes ainsi que le nombre de développeurs persistants.

Volatilité des développeurs

Nous avons choisi dans la section 5.5 une taille de six mois pour les périodes p_1 et p_2. Pour observer l'évolution du *turnover*, nous utilisons une fenêtre glissante pour ces deux périodes, en commençant avec un commit c_i se situant 12 mois après le début de l'historique *Git* du projet (c_{i-1} est donc six mois après le début de l'historique et c_{i-2} est le premier commit de l'historique), puis en faisant glisser les *commits* avec un pas de deux semaines jusqu'à atteindre le dernier commit disponible dans l'historique (tout en restant sur la même branche du gestionnaire de versions). Les figures 5.3 et 5.4 présentent les résultats obtenus. La figure 5.3 présente le nombre de développeurs du projet et les nombres de nouveaux arrivants, développeurs sortants et développeurs persistants. La figure 5.4 présentent ces même chiffres sous forme de pourcentage du nombre de développeurs du projet.

Nous pouvons observer deux phases lors de l'évolution des projets de notre corpus. La première phase où le nombre de développeurs croit fortement, avec une proportion de nouveaux arrivants très élevée, est visible au début des projets Angular.JS et Ansible. Ces deux projets sont les seuls de notre corpus ayant un historique de développement entièrement enregistré avec Git, ce qui explique que le nombre de développeurs soit initialement élevé pour les autres projets. Tous les projets entrent ensuite dans une phase où les nombres de nouveaux arrivants et de développeurs sortants sont similaires, ce qui entraine une stabilisation du nombre total de développeurs.

Synthèse 5.6.1.1
Dans l'ensemble des projets observés le nombre de nouveaux arrivants ou développeurs sortants est très élevé. Pour un point donné de l'historique de ces projets, la proportion de développeurs persistants atteint au plus 20% (40% pour Mono). Ces observations sont en accord avec nos attentes : le *turnover* dans les projets *open-source* est grand et seule une minorité de développeurs contribue à ces projets pendant plus de six mois.

Taux de conversion et motivations des développeurs persistants

Étant donné que les développeurs persistants sont relativement rares, nous nous sommes intéressés plus en détails à ceux-ci et à leurs particularités. Si l'on regarde le taux de conversion des nouveaux arrivants en développeurs persistants on s'aperçoit que seulement 8% à 21% des nouveaux arrivants deviendront un jour des développeurs persistants (voir table 5.1), à l'exception du projet Mono où les nouveaux arrivants sont plus impliqués dans le projet, avec un taux de conversion de 38%.

Tableau 5.1 : Taux de conversion des nouveaux arrivants en développeurs persistants.

Angular.JS	Ansible	Jenkins	JQuery	Mono	PHPUnit	Rails
8%	16%	21%	19%	38%	7%	17%

Afin de mieux comprendre ce qui fait qu'un nouvel arrivant deviendra développeur persistant dans les projets de notre corpus, nous avons regardé manuellement les profils des dix développeurs ayant le plus de *commits* dans chacun des projets. Nous avons cherché manuellement sur leurs profils GitHub, LinkedIn et sur leurs pages personnelles des informations relatives à leur motivations. Nous en avons déduit quatre catégories de développeurs :

- les individus qui sont rémunérés par une entreprise sponsorisant le projet. Par exemple sept des dix développeurs les plus actifs du projet Angular.JS sont des salariés de Google, qui est le sponsor principal du projet ;
- les individus qui sont rémunérés par une entreprise développant le projet avec un intérêt financier immédiat. C'est le cas notamment du projet Ansible, qui est un projet open-source faisant partie d'une suite logicielle propriétaires et payante, développée par Ansible, Inc. ;
- les individus qui sont consultants pour une technologie particulière, comme c'est le cas pour cinq des dix développeurs principaux de Rails ;
- les individus contribuant pendant leur temps libre, sans intérêt financier perceptible. Seuls deux profils parmi les 50 que nous avons observés appartiennent à cette catégorie.

Ces quatre catégories sont en accord avec d'autres taxonomies, comme par exemple celle de Berdou, qui a identifié différents types de développeurs contribuant aux projets *open-source* via des entretiens avec ces derniers [Berdou, 2006].

5.6.2 Répartition des acteurs du *turnover* entre les modules (PR2)

Les visualisations présentées en figure 5.5 présentent les valeurs des métriques de *turnover* pour chacun des modules des projets de notre corpus. Chaque ligne de ces visualisations correspond à un module et chaque colonne à une métrique. Plus la couleur d'une

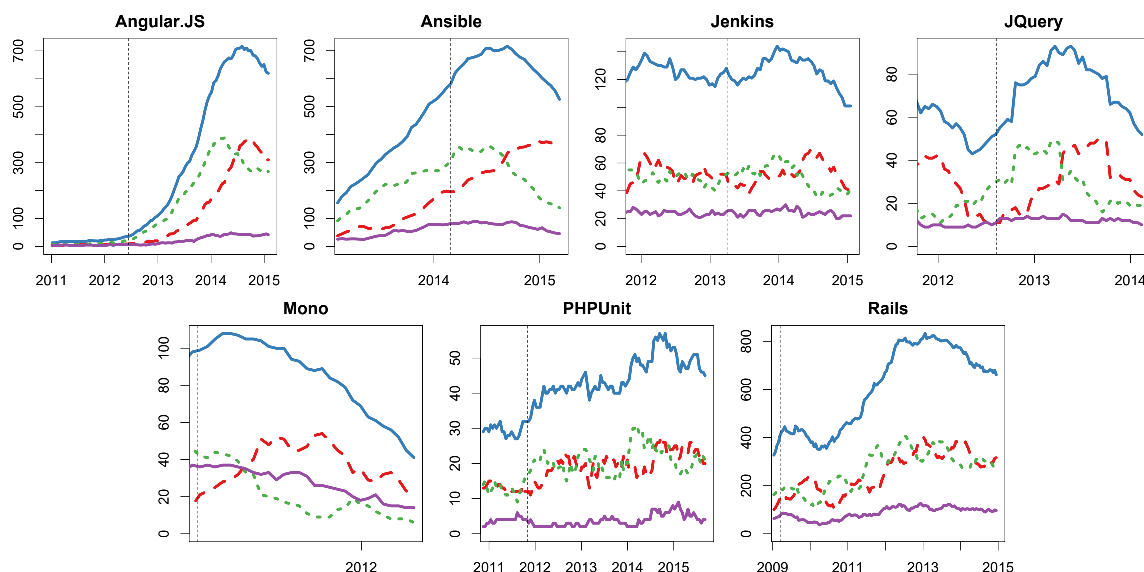

FIGURE 5.3 : Évolution du *turnover*. La ligne continue bleue (en haut) représente le nombre total de développeurs, la ligne continue violette (en bas) le nombre de développeurs persistants, la ligne pointillée verte le nombre de nouveaux arrivants, et la ligne à tirets rouge le nombre de développeurs sortants.

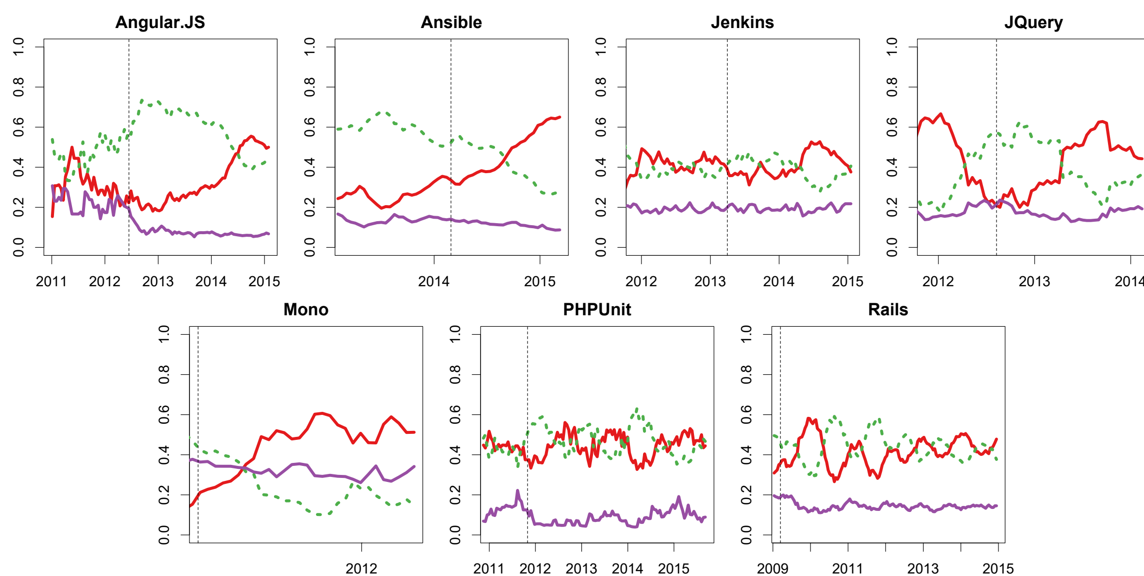

FIGURE 5.4 : Évolution du *turnover*. Cette figure présente les ratios de nouveaux arrivants, développeurs sortants et développeurs persistants. La ligne continue violette représente le ratio de développeurs persistants, la ligne pointillée verte le ratio de nouveaux arrivants, et la ligne rouge le ratio de développeurs sortants.

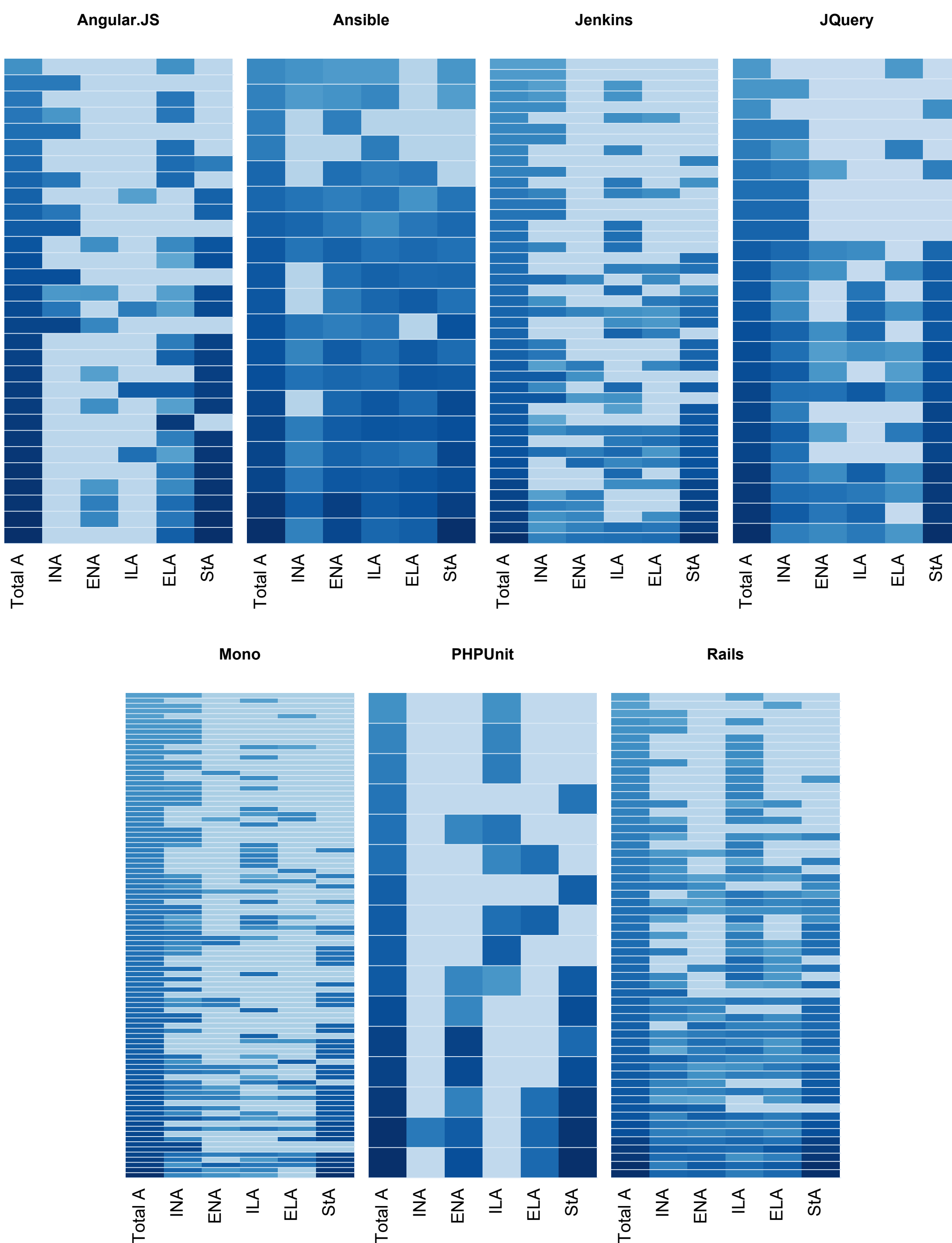

FIGURE 5.5 : Visualisation de l'activité des acteurs du *turnover*. Chaque rangée de la matrice correspond à un module. Plus la couleur d'une cellule est foncée, plus l'ensemble de développeurs correspondants est actif sur ce module.

case est foncée, plus la valeur de la métrique pour le module en question est élevée. Les métriques ont été calculées pour la version v_i de chacun des projets, avec des périodes p_1 et p_2 d'une durée de six mois.

Dans le projet Angular.JS, la plus grande partie de l'activité est due aux développeurs persistants ou aux développeurs sortants externes. Le niveau d'activité élevé des développeurs sortants externes est principalement dû aux contributions d'un individu, un développeur majeur du projet qui a été inactif pendant les six mois précédant la sortie de la version 1.0.0 du projet.

Dans le projet Ansible, nous remarquons que toutes les catégories d'acteurs du *turnover* ont un niveau d'activité similaire et que les développeurs contribuent à un ensemble plus grand de modules. Cela diffère des autres projets, en particulier pour les nouveaux arrivants externes qui sont généralement peu actifs : ici, seuls quatre modules sur les 19 n'ont pas reçu de contributions de nouveaux arrivants externes. Nous nous sommes donc intéressés plus en détails au module où les nouveaux arrivants externes étaient les plus actifs, qui correspond au module contenant les composants liés à l'intégration d'Ansible dans le *cloud*. Parmi les nouveaux arrivants ayant réalisé le plus de contributions dans ce module, un d'entre eux est un employé de Ansible, Inc. et deux sont employés par une entreprise nommée Rackspace et ont consacré la plupart de leurs contributions au projet au développement d'un composant permettant l'intégration d'Ansible dans la plateforme de *cloud* développée par Rackspace. Ces développeurs ont très probablement été rémunérés pour réaliser ces contributions, ce qui explique leur niveau d'activité élevé qui n'est généralement pas visible pour les nouveaux arrivants.

Le projet PHPUnit contient très peu de contributions venant de nouveaux arrivants internes, avec seulement un module affecté par leur activité. Les nouveaux arrivants externes sont cependant relativement actifs, une partie importante de cette activité étant dûe à un seul individu, employé au moment de ces contributions par une entreprise développant des applications PHP.

L'activité des développeurs dans les projets restants (Jenkins, JQuery, Mono et Rails) suit des schémas similaires :

- les nouveaux arrivants et développeurs sortants internes sont actifs sur une majorité de modules ;
- les contributions des nouveaux arrivants et des développeurs sortants externes sont restreintes à un faible nombre de modules, avec un niveau d'activité restant relativement faible ;

Synthèse 5.6.2.2

Si l'on considère l'ensemble des projets étudiés, il apparait que le *turnover* est présent pour toutes les parties du code source et seuls quelques modules ne reçoivent des contributions que de développeurs persistants. Nous avons observé manuellement quelques cas où l'activité des nouveaux arrivants est anormalement élevée et ces cas coïncident avec la présence de développeurs rémunérés pour contribuer au projet.

5.6.3 Relation avec la fiabilité logicielle (PR3)

Pour répondre à notre troisième problématique de recherche, liée à la relation entre le *turnover* et la fiabilité logicielle, nous utilisons les mesures de fiabilité extraites grâce à la méthodologie décrite dans le chapitre 3. La mesure de qualité que nous utilisons ici est la densité de *commits* correctifs par ligne de code.

Afin d'évaluer la relation entre chacune de nos mesures de *turnover* et la densité de *commits* correctifs, nous utilisons le coefficient de corrélation de Spearman, associé à la méthode *bootstrap* permettant de fournir des intervalles de confiance pour ce coefficient de corrélation. La figure 5.6 présente les corrélations obtenues pour chaque métrique et chaque projet de notre corpus.

Turnover interne

Le *turnover* interne est représenté par les deux métriques *INA* et *ILA* et les corrélations de ces métriques avec la fiabilité sont illustrées sur la première rangée de la figure 5.6.

Il apparait que l'activité des nouveaux arrivants internes est corrélée positivement avec la densité de *commits* correctifs dans seulement quatre projets sur sept. De plus, cette corrélation est relativement faible dans la plupart des cas, avec des intervalles de confiance qui tendent à rester dans des valeurs inférieures à 0.50.

L'activité des développeurs sortants internes ne présente pas de corrélation significative et cohérente entre les projets. Seuls deux intervalles de confiances sont strictement positifs ou négatifs :

- pour JQuery, il existe une corrélation moyenne à forte entre l'activité des développeurs sortants internes et la fiabilité, cette corrélation étant positive ;
- pour PHPUnit, cette corrélation est également moyenne à forte mais est négative, c'est-à-dire que plus le nombre de développeurs sortants internes est élevé, moins la densité de *commits* correctifs l'est.

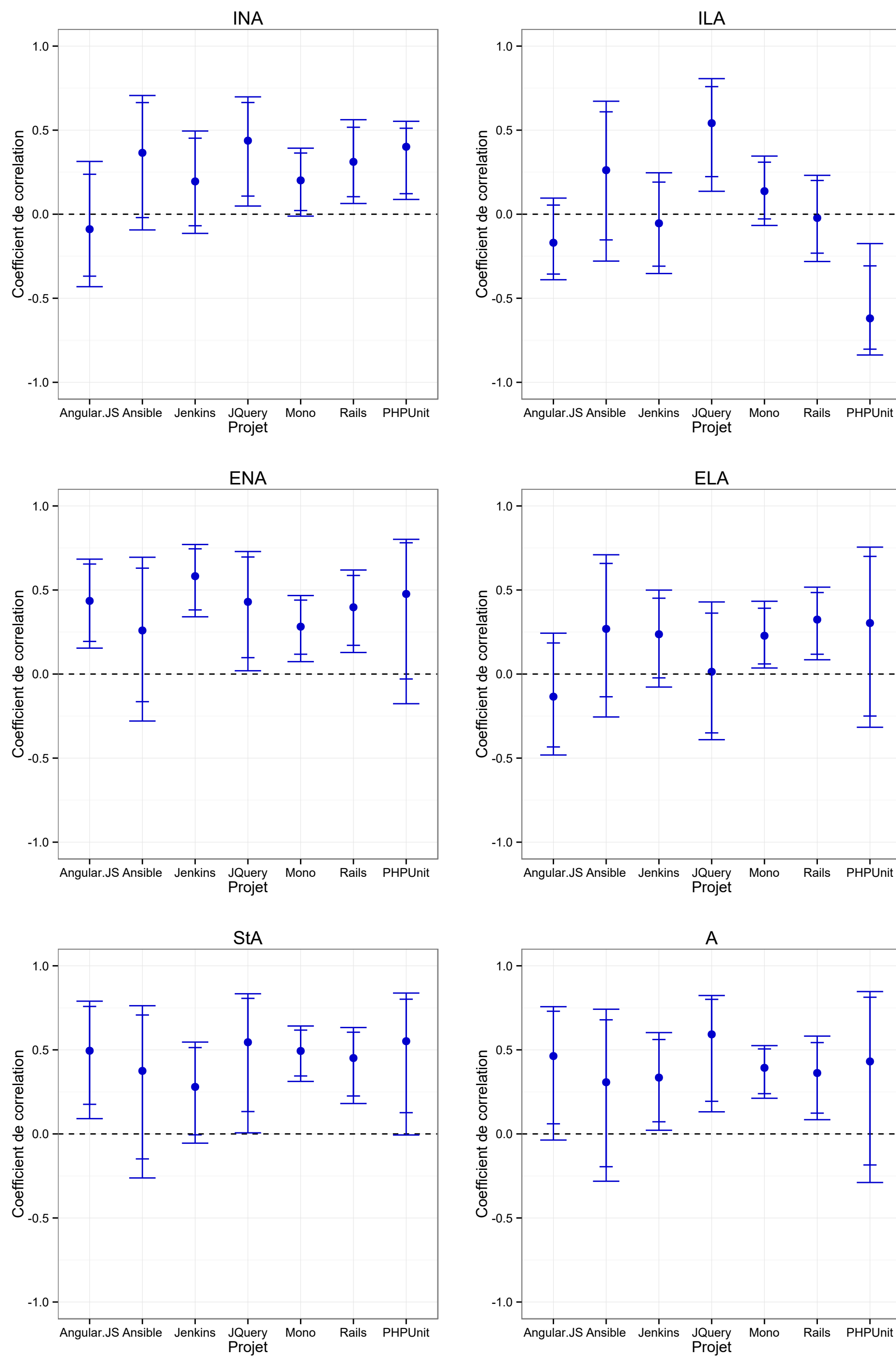

FIGURE 5.6 : Corrélation entre les métriques de rotation des effectifs et la densité de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*.

Synthèse 5.6.3.3

Ces résultats ne permettent pas de confirmer les théories évoquées concernant l'impact positif du *turnover* interne. En revanche, le fait que les nouveaux arrivants internes soient liés à une densité plus élevée de *commits* correctifs est en accord avec une partie des résultats de Rahman et Devanbu, suggérant que l'expérience des développeurs sur un module est plus importante que l'expérience sur le projet en ce qui concerne la qualité de ses contributions [Rahman et Devanbu, 2011].

***Turnover* externe**

Le *turnover* externe est représenté par les métriques *ENA* et *ELA* et les corrélations de ces métriques avec la fiabilité sont illustrées sur la seconde rangée de la figure 5.6.

L'activité des nouveaux arrivants externes est corrélée dans cinq des sept projets avec une corrélation positive et avec des intervalles de confiance à 95% strictement positifs. Dans le cas de PHPUnit, bien que l'intervalle de confiance à 95% ne soit pas strictement positif, la borne inférieure de l'intervalle à 90% est de l'ordre de -0.01 et il est également possible de considérer la corrélation comme positive. L'intervalle de confiance pour le projet *Ansible* a une partie négative non négligeable, ce qui ne nous permet pas de considérer cette corrélation comme positive. En ce qui concerne la force de cette corrélation, elle varie légèrement entre les projets, allant de faible à moyenne pour le projet Mono, et de moyenne à forte pour le projet Jenkins.

Synthèse 5.6.3.4

L'activité des développeurs sortants externes n'est positive que dans deux projets, Mono et Rails, voire trois projets si l'on considère l'intervalle de confiance à 90%, qui est strictement positive pour Jenkins. Dans ces trois projets, la corrélation est au mieux moyenne. Les résultats relatifs à l'activité des nouveaux arrivants externes sont en accord avec la théorie concernant leur inexpérience. La corrélation entre leur activité et la fiabilité est positive et peut être forte dans certains cas, ce qui est un résultat significatif lorsque l'on prend en compte le fait que cette catégorie de développeurs est généralement la moins active. L'impact des développeurs sortants n'est en revanche pas constant entre les projets, ce qui ne permet pas de confirmer les observations de Mockus [Mockus, 2010].

Développeurs persistants et activité globale

Enfin, nos deux dernières métriques concernent l'activité des développeurs persistants et l'activité de tous les développeurs. Les corrélations de ces métriques avec la fiabilité sont illustrées sur la troisième rangée de la figure 5.6.

Il apparaît que les résultats de ces deux métriques présentent de fortes similarités. Ces similarités sont explicables par le fait que la majorité des contributions est apportée par des développeurs persistants et que l'activité des développeurs persistants est donc très proche de l'activité globale. Ces deux métriques sont corrélées positivement avec la fiabilité logicielle. L'activité globale sur le projet est généralement corrélée avec la fiabilité, étant donné que les modules étant les plus susceptibles de subir l'introduction de bogues sont ceux qui sont modifiés.

Synthèse 5.6.3.5
Une donnée importante concernant ces corrélations est que, à l'exception de JQuery, la corrélation obtenue avec l'activité globale est surpassée par la corrélation obtenue avec l'activité des développeurs persistants ou des nouveaux arrivants externes. Cela suggère que ces deux métriques peuvent être de meilleurs indicateurs de qualité que l'activité globale des développeurs.

5.6.4 Importance relative des métriques

En accord avec notre méthodologie définie chapitre 3, nous appliquons la méthode PMVD afin de déterminer l'importance des métriques de *turnover* dans l'explication de la fiabilité des modules logiciels. Nous n'incluons dans cette analyse que les métriques ayant présenté une corrélation avec la fiabilité, à savoir l'activité des nouveaux arrivants externes et internes (ENA et INA) et l'activité des développeurs persistants (StA). Nous ajoutons à ces métriques la taille des modules logiciels, le nombre de développeurs moyen entre les deux périodes p_1 et p_2 et l'activité globale des développeurs.

Étant donné que la méthode PMVD se base sur une régression linéaire, nous utilisons le nombre de *commits* correctifs (au lieu de la densité), qui est plus approprié. La figure 5.7 présente l'importance relative de ces métriques dans chacun des projets.

Il apparait que, à l'exception de Mono et Rails, une ou plusieurs de nos métriques de *turnover* sont importantes dans le modèle multivarié. Les trois métriques apportent, à tour de rôle, des contributions non négligeables à la précision des modèles (le R^2) et dans trois des projets l'activité des développeurs persistants fait partie des deux métriques les plus importantes. L'importance de l'activité des nouveaux arrivants externes ou internes est moins élevée, mais peut attendre 10% du R^2 dans certains cas.

Synthèse 5.6.4.6
Les métriques liées à l'activité des nouveaux arrivants externes ou internes, ainsi que l'activité des développeurs persistants, sont des métriques qui sont donc susceptibles d'apporter une plus-value lors de la création de modèles de prédiction de bogues.

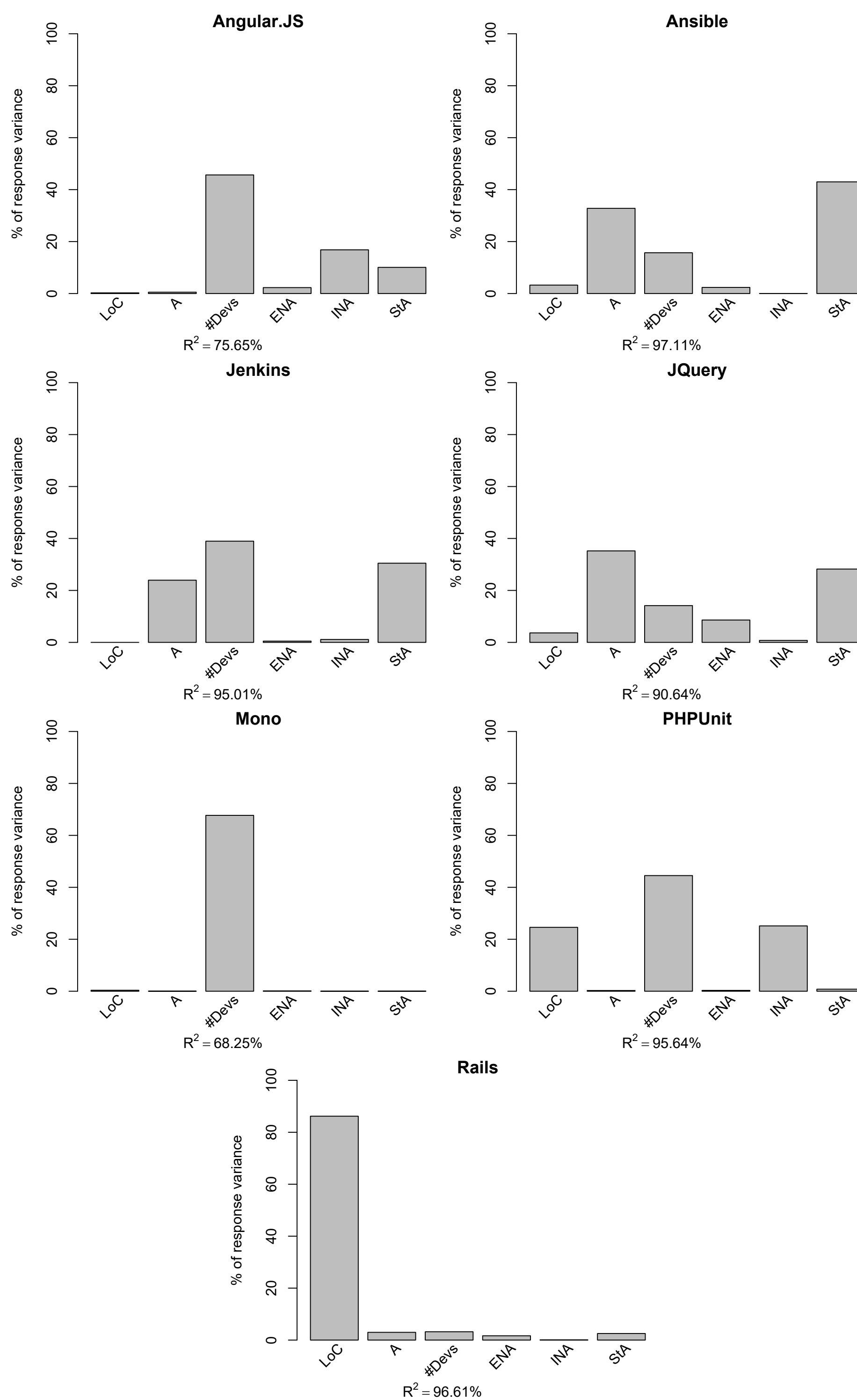

FIGURE 5.7 : Importance relative des métriques dans un modèle multivarié avec pour variable dépendante le nombre de *commits* correctifs.

5.7 Validité de l'étude

La validité de notre étude est principalement liée à la validité du jeu de données, que nous abordons dans le chapitre 3.

La généralisation des résultats présentés dans cette étude est notre intérêt premier. Bien que le jeu de données que nous avons construit regroupe des projets développés dans différents langages de programmation par des centaines de développeurs, cette étude n'a été réalisée que sur sept projets sélectionnés manuellement. Afin de surmonter cette limitation, cette étude doit donc être reproduite sur de nouveaux projets dans le but de confirmer et d'améliorer les résultats présentés dans cette étude. Une barrière pour accomplir cet objectif est la construction d'un jeu de données fiable, comme nous l'avons vu dans le chapitre 3.

Les données extraites du projet Mono ne satisfont pas complètement une des exigences de notre méthodologie, à savoir le fait que l'activité des développeurs soit extraite depuis l'historique d'un projet Git. Sur les six mois de la période p_1, quatre proviennent d'un dépôt Subversion, qui ne garantit pas une identification précise des contributeurs. Afin d'évaluer l'impact possible de cette limitation, nous avons parcouru manuellement l'historique de ces quatre premiers mois à la recherche de mentions de contributeurs autres que les auteurs des *commits* Subversion et nous n'avons trouvé aucune mention de contributeurs ayant soumis un *patch*.

5.8 Limites et travaux futurs

Les résultats présentés dans ce chapitre permettent d'apporter un premier éclairage sur le *turnover* et la fiabilité logicielle. Cette première étude concernant le *turnover* dans le contexte *open-source* est cependant limitée par plusieurs facteurs et laisse entrevoir de nombreux travaux futurs.

Nos métriques sont limitées par le fait qu'elles ne considèrent que l'activité ayant lieu sur le gestionnaire de versions. Cependant, un projet logiciel contient d'autres types de dépôts logiciels tels que les listes de diffusion et les systèmes de suivi de bogues. Si l'on prend en compte l'activité générée dans ces dépôts logiciels, certains développeurs considérés comme externes au projet pourraient se révéler être des contributeurs persistants du projet. Nos travaux futurs incluent l'étude du *turnover* via des métriques se basant sur de multiples dépôts logiciels.

Lors de l'étude des schémas de contributions sur les différents modules, nous avons remarqué des différences dans les motivations des développeurs qui impactent leur niveau de contribution dans le projet. Nous aimerions étendre ces observations en étudiant la liste complète des développeurs de chacun des projets et en les séparant en différentes catégories liées à leurs motivations.

Les métriques que nous avons définies n'ont pas été validées par les développeurs des projets étudiés. Une étude comprenant des questionnaires ou des entretiens avec ces développeurs permettrait de valider ou d'améliorer ces métriques, tout en permettant une compréhension approfondie des résultats observés.

Enfin, l'analyse statistique réalisée dans ce chapitre considère une relation monotone entre les métriques de *turnover* et la fiabilité logicielle. L'état de l'art que nous avons réalisé dans le chapitre 2 suggère cependant que cette relation n'est pas monotone, et qu'une absence de *turnover* serait dommageable pour la fiabilité d'un logiciel. Nous aimerions donc tester cette relation à l'aide d'outils statistiques plus puissants.

CHAPITRE

6 Conclusion

6.1 Résumé des contributions

La fiabilité logicielle est un des objectifs principaux du génie logiciel et la recherche d'indicateurs permettant d'estimer et de prévoir cette fiabilité est le sujet de nombreux travaux. À titre d'exemple, plus de 200 études ont été publiées entre janvier 2000 et décembre 2010 avec pour objectif la prédiction de bogues dans le code source [Hall *et al.*, 2012]. Les métriques utilisées pour tenter de réaliser ces prédictions peuvent se baser sur le design du code source, les bogues précédents, la nature des modifications effectuées sur le code ou les développeurs responsables de ces modifications. Dans le domaine de recherche appelé fouille de données dans les dépôts logiciels (ou *mining software repositories* en anglais), ces métriques sont extraites depuis l'historique de développement des projets, qui est stocké notamment dans des gestionnaires de versions. Les études réalisées à partir de ces données sont des études empiriques, c'est-à-dire que leur but est de valider des théories par l'accumulation d'observations statistiquement significatives.

Lors de la réalisation de cette thèse, nous avons choisi de nous intéresser aux métriques de procédé dans les projets *open-source*, avec en particulier des métriques centrées sur l'organisation des développeurs. Les projets *open-source* reçoivent la participation de nombreux contributeurs et ont une organisation différente des projets dits *industriels*, avec un cœur de développeurs fortement impliqués dans le projet et une grande part de contributeurs apportant seulement une correction ou une fonctionnalité donnée. Nous avons validé différentes métriques de procédé en observant leur relation avec la fiabilité sur un jeu de données que nous avons créé.

Méthodologie. Notre première contribution, présentée dans le chapitre 3, est la définition d'une méthodologie permettant la réalisation de telles études, en suivant les diffé-

rentes étapes des procédés de fouille de données dans les dépôts logiciels et permettant la reproduction nécessaire à la validation de toute étude empirique. Les différentes étapes définies par notre méthodologie sont les suivantes :

1. la sélection des projets permettant l'obtention de données précises. Les critères de sélection portent sur la nature du gestionnaire de versions utilisé par le projet, le processus de production et de maintenance utilisé par les développeurs et des attributs tels que le langage de programmation et la taille des projets ;
2. l'extraction des métriques d'organisation des développeurs et de fiabilité depuis l'historique de développement. Nous décrivons les différentes exigences devant être respectées par le procédé d'extraction afin d'obtenir des mesures précises et les solutions que nous avons choisies. Nous décrivons en particulier le procédé utilisé pour le découpage du code source en modules, l'algorithme utilisé pour la détection des alias multiples des développeurs et le procédé d'extraction des *commits* correctifs ;
3. l'analyse statistique des métriques, reposant sur des statistiques telles que la corrélation de Spearman avec des intervalles de confiance calculées avec la méthode *bootstrap* et la mesure de l'importance relative des métriques.

Enfin, nous présentons brièvement les projets sélectionnés par notre méthodologie.

Répartition des contributions. Notre seconde contribution, présentée dans le chapitre 4, a pour but d'évaluer la relation entre la répartition des contributions et la fiabilité logicielle. L'étude menée dans ce chapitre est une reproduction d'études précédentes où nous réutilisons les métriques définies par Bird *et al.* [Bird *et al.*, 2011] et par Posnett *et al.* [Posnett *et al.*, 2013].

Les métriques définies par Bird *et al.*, que ces derniers ont étudiées sur des projets développés chez Microsoft, reposent sur le concept de propriété du code (*code ownership* en anglais) et pour chaque module logiciel, séparent les développeurs en deux catégories : les contributeurs majeurs et les contributeurs mineurs [Bird *et al.*, 2011]. Nos résultats relatifs à la relation entre ces métriques et la fiabilité confirment les résultats de Bird *et al.* : le nombre de développeurs mineurs est corrélé positivement avec le nombre de *commits* correctifs réalisés pour un module, c'est-à-dire que plus le nombre de développeurs mineurs augmente sur un module, plus sa fiabilité diminue. À l'inverse, une propriété forte du code d'un module, c'est-à-dire la présence d'un développeur ayant réalisé une forte majorité de contributions, augmente la fiabilité d'un module. L'intérêt des métriques de propriété du code dans les projets *open-source* est cependant mitigé lorsqu'on compare ces métriques à d'autres métriques de base telles que la taille des modules, le nombre de développeurs ou le *code churn* du module. Il apparait que dans les projets de notre corpus, le nombre de développeurs mineurs (la métrique *Minor*) est colinéaire avec le nombre de développeurs total d'un module, ce qui rend cette métrique redondante avec une métrique existante. Les deux autres métriques, à savoir le nombre de développeurs majeurs (la métrique *Major*) et le ratio de contributions réalisées par le développeur principal d'un module (la métrique

MVO) ont une importance faible voire nulle lorsqu'on les ajoute à un modèle de régression linéaire multiple contenant les métriques de base énoncées ci-dessus.

La métrique définie par Posnett *et al.*, que ces derniers ont étudiée sur des projets *open-source* de la fondation Apache, mesure la concentration de l'activité des développeurs sur un module par le bais de la métrique MAF qui est identique à une mesure utilisée en écologie pour évaluer l'interaction entre différentes espèces animales ou végétales [Posnett *et al.*, 2013; Blüthgen *et al.*, 2006]. Notre hypothèse concernant cette métrique est qu'elle est corrélée négativement avec le nombre et la densité de *commits* correctifs, c'est-à-dire que plus l'activité des développeurs est concentrée, plus la fiabilité augmente. Nous n'avons pas pu valider cette hypothèse, étant donné que peu de corrélations obtenues sont significatives et que les corrélations observées ne sont pas cohérentes entre les projets.

Turnover. Notre troisième contribution, présentée dans le chapitre 5, est la réalisation d'une étude relative à l'impact du *turnover* des développeurs sur la fiabilité. Ce type d'étude étant, à notre connaissance, inédite avec des projets *open-source*, nous avons défini les métriques permettant la quantification du *turnover* externe et interne dans ces projets. Nous avons défini cinq catégories d'acteurs du *turnover*, à savoir les nouveaux arrivants internes et externes, des développeurs sortants internes et externes et les développeurs persistants. Pour chacune de ces catégories, nous avons défini une métrique basée sur l'activité des développeurs de cette catégorie.

Nous avons observé les niveaux de *turnover* présents à l'échelle des projets et des modules de notre corpus et identifié différents schémas d'activité des acteurs du *turnover*. Lorsque l'on regarde la relation entre ces métriques et la densité de *commits* correctifs, il apparait que :

- les nouveaux arrivants externes ont un impact négatif sur la fiabilité, confirmant les théories relatives à leur inexpérience et au temps d'adaptation qui leur est nécessaire ;
- l'activité des développeurs persistants est également corrélée avec la densité de *commits* correctifs ;
- l'activité des nouveaux arrivants internes au projet est corrélée positivement avec la densité de *commits* correctifs, bien que cette corrélation soit moins forte que pour les deux métriques ci-dessus.

Nous avons mesuré l'importance de ces trois métriques dans un modèle multivarié contenant des métriques de taille des modules, d'activité de l'ensemble des développeurs et de nombre de développeurs : au moins une des métriques de *turnover* est relativement importante dans cinq des sept projets étudiés, ce qui confirme l'intérêt de ces métriques pour développer des modèles de prédiction de la fiabilité.

6.2 Perspectives

Nous abordons ici les différentes perspectives qui permettraient d'étendre les travaux présentés dans cette thèse.

6.2.1 Motivations des développeurs

Les métriques utilisées dans nos travaux considèrent que les développeurs sont égaux en termes de motivations et d'implication dans la qualité des contributions, ce qui n'est pas nécessairement le cas. Au vu du succès des projets *open-source* au fil des années, ce modèle a attiré la participation de nombreuses entreprises [Crowston *et al.*, 2008]. De ce fait, certains développeurs reçoivent un salaire lorsqu'ils réalisent ces contributions, alors que d'autres agissent en bénévoles [Riehle *et al.*, 2014] et de nombreux projets *open-source* sont sponsorisés voire gérés par des entreprises [Capiluppi *et al.*, 2012].

Lors de notre étude des schémas de *turnover*, nous avons observé que les motivations des développeurs pouvaient avoir un impact sur leur niveau de contributions, avec la présence de nouveaux arrivants externes très actifs dans le projet Ansible. Il est possible de définir de nouvelles catégories de développeurs, tels que les développeurs bénévoles et les développeurs salariés, avec éventuellement différentes catégories de développeurs salariés en accord avec la taxonomie de Berdou, qui a réalisé des entretiens avec des développeurs salariés de projets *open-source* afin de comprendre leurs motivations [Berdou, 2006]. Ces nouvelles catégories permettraient de mieux comprendre l'impact de l'organisation des développeurs, notamment dans le cas du *turnover*, où il est possible que l'arrivée ou le départ d'un développeur ait un impact différent selon ses motivations et son statut de salarié ou bénévole. De plus, il serait possible de tester l'impact de ces nouvelles catégories de développeurs sur la fiabilité, ce qui n'a été le sujet que d'un petit nombre d'études [Sato *et al.*, 2013; Duc *et al.*, 2011].

L'assignation des développeurs aux différentes catégories requiert cependant une grande part de recherche manuelle sur des pages personnelles des développeurs, ce qui amène des difficultés tant en termes de coût de construction du jeu de données qu'en termes de possibilité de reproduction de l'étude, étant donnée que la permission des développeurs serait requise pour permettre le partage de ce jeu de données.

6.2.2 Mesure précise de l'activité

La mesure des phénomènes décrits dans nos problématiques repose en partie sur l'évaluation du niveau de contribution de chaque développeur au code source. La mesure de cette activité peut être réalisée à différents degrés de précision : le niveau de granularité le plus élevé consiste à mesurer le nombre de *commits* affectant un module logiciel, sans tenir compte du contenu des *commits*. Dans notre thèse, cette mesure utilise le nombre de lignes de code ajoutées, modifiées ou supprimées dans chaque *commit* (mesure appelée

code churn en anglais) [Munson et Elbaum, 1998], ce qui permet de distinguer les modifications affectant une grande partie du contenu des fichiers de celles qui sont localisées sur un faible nombre de lignes.

Cependant, ces mesures sont limitées dans leur précision par le fait qu'elles ne prennent pas en compte la syntaxe des langages de programmation utilisés, ce qui est possible via des outils tels que GumTree [Falleri *et al.*, 2014]. Certaines modifications, telles que l'édition de la liste des licences en tête de fichier, peuvent affecter un grand nombre de lignes de code, sans toutefois avoir un impact sur le comportement du logiciel [Hindle *et al.*, 2008]. L'impact de ce choix, c'est-à-dire la différence entre les valeurs des métriques utilisant le nombre de lignes de code et de celles utilisant la syntaxe du langage, n'est pas connu. Outre la possibilité de mesurer la quantité de modifications avec une plus grande précision, avoir une analyse syntaxique des changements introduits dans chaque *commit* permettrait de décomposer l'activité des développeurs en différents types d'actions, telles que la documentation, le renommage de variable, etc. La prise en compte de la syntaxe du langage permettrait également de filtrer les modifications non-essentielles, comme l'ont montré Kawrykow et Robillard [Kawrykow et Robillard, 2011].

La question se pose alors de l'impact de la mesure d'activité utilisée sur les métriques liées à la distribution des contributions et au *turnover*, que nous utilisons dans les deux problématiques de cette thèse. Certains types d'actions réalisées par les développeurs pourraient affecter plus que d'autres la fiabilité du logiciel et les types d'actions n'ayant pas d'effet sur la fiabilité introduiraient alors du bruit dans les données extraites. Outre le fait de proposer des indicateurs de qualité plus précis pour les gestionnaires de projets, répondre à ces problématiques permettrait de donner des indications aux chercheurs en génie logiciel sur l'intérêt de considérer la syntaxe du langage de programmation lorsque l'on considère l'activité des développeurs.

La réalisation de cette perspective peut être faite via l'outil GumTree [Falleri *et al.*, 2014], mais requiert l'intégration des *parsers* des différents langages de programmation de notre jeu de données. De plus, nos études préliminaires ont montré qu'un grand nombre d'actions différentes étaient possibles lorsqu'on considère un grain aussi fin que l'arbre de syntaxe abstraite du code et qu'une analyse approfondie est nécessaire afin de définir les différentes catégories d'actions possibles sur le code source.

ANNEXE

A Métriques de *turnover* : sélection des périodes

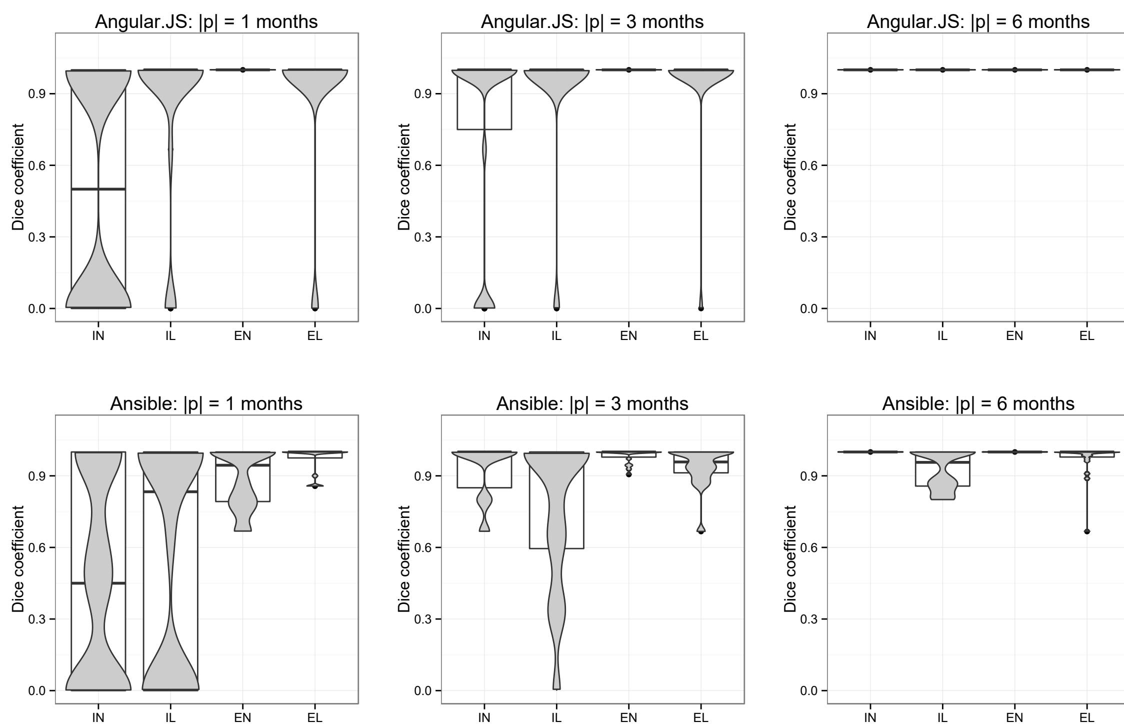

FIGURE A.1 : Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (1 sur 3).

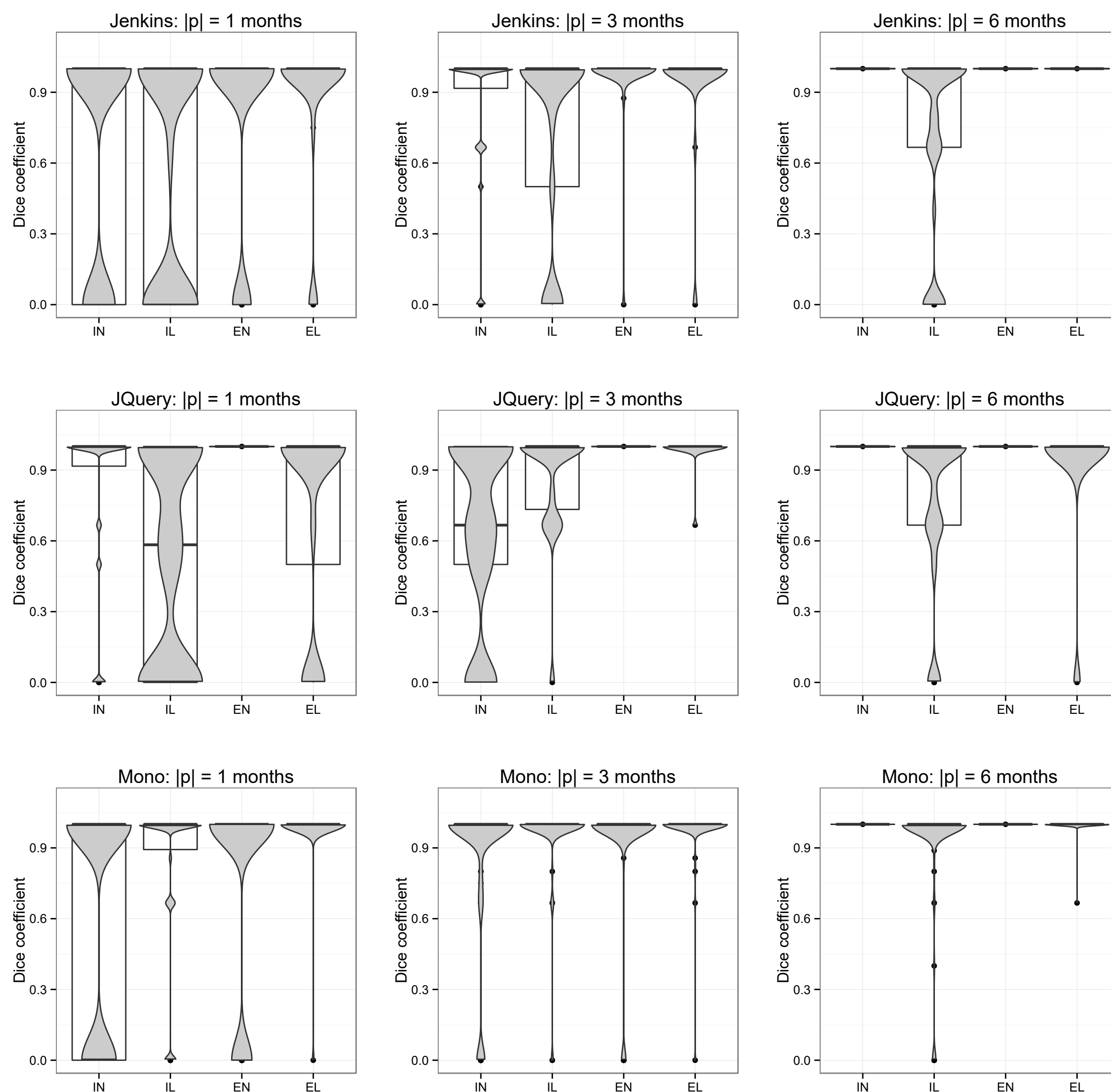

FIGURE A.2 : Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (2 sur 3).

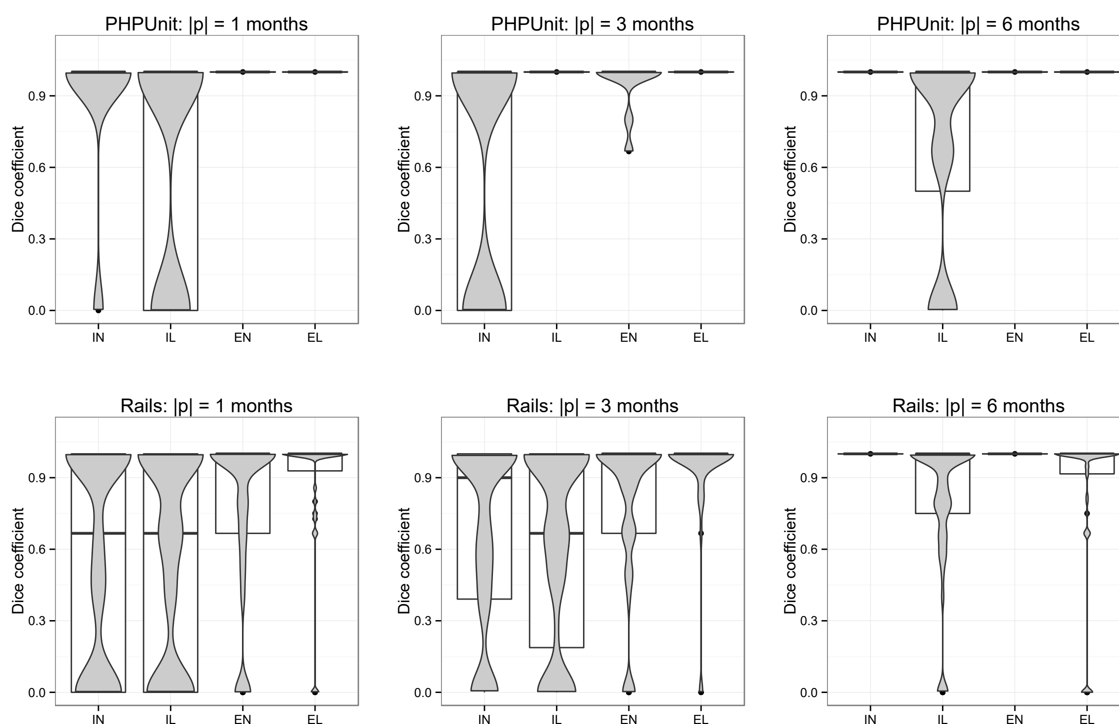

FIGURE A.3 : Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (3 sur 3).

Bibliographie

Argote, L. et Epple, D. (1990). Learning curves in manufacturing. *Science*, 247(4945):920–924. Cité page 65.

Asundi, J. et Jayant, R. (2007). Patch review processes in open source software development communities : A comparative case study. *In 40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007.* Cité page 15.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P. et Bernstein, A. (2010). The missing links : bugs and bug-fix commits. *In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, page 97–106. Cité pages 13 et 34.

Beck, F. et Diehl, S. (2012). On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004. Cité page 32.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77. Cité pages 2 et 52.

Beck, K. (2003). *Test-driven development by example*. Addison-Wesley Professional. Cité page 2.

Berdou, E. (2006). Insiders and outsiders : paid contributors and the dynamics of cooperation in community led F/OS projects. *In Open Source Systems*, page 201–208. Springer. Cité pages 74 et 88.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V. et Devanbu, P. (2009a). Fair and balanced ? : Bias in bug-fix datasets. *In 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, ESEC/FSE ’09, page 121–130. Cité pages 6, 12 et 46.

Bird, C., Gourley, A. et Devanbu, P. (2007). Detecting patch submission and acceptance in OSS projects. *In Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 26–. IEEE Computer Society. Cité page 15.

Bird, C., Nagappan, N., Murphy, B., Gall, H. et Devanbu, P. (2011). Don't touch my code ! : examining the effects of ownership on software quality. *In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, page 4–14. ACM. Cité pages 6, 7, 16, 20, 48, 50, 52, 56, 59, 62 et 86.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M. et Devanbu, P. (2009b). The promises and perils of mining git. *In 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09. Cité pages 6 et 15.

Bissyandé, T. F., Thung, F., Wang, S., Lo, D., Jiang, L. et Réveillère, L. (2013). Empirical evaluation of bug linking. *In Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 1–10. Cité page 13.

Blüthgen, N., Menzel, F. et Blüthgen, N. (2006). Measuring specialization in species interaction networks. *BMC Ecology*, 6(1):9. Cité pages 19, 51 et 87.

Brito e Abreu, F. et Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. *In Proceedings of the 3rd International Software Metrics Symposium*. Cité page 3.

Brooks, F. P. (1975). *The mythical man-month*, volume 1995. Addison-Wesley Reading. Cité pages 7 et 17.

Canfora, G., Di Penta, M., Oliveto, R. et Panichella, S. (2012). Who is going to mentor newcomers in open source projects ? *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, page 44. Cité pages 2 et 65.

Canty, A. et Ripley, B. D. (2013). *boot : Bootstrap R (S-Plus) Functions*. R package version 1.3-9. Cité page 40.

Capiluppi, A. (2013). Similarities, challenges and opportunities of wikipedia content and open source projects. *Journal of Software : Evolution and Process*, 25(9):891–914. Cité page 21.

Capiluppi, A. et Adams, P. J. (2009). Reassessing brooks' law for the free software community. *In Open Source Ecosystems : Diverse Communities Interacting*, page 274–283. Springer. Cité page 18.

Capiluppi, A., Stol, K.-J. et Boldyreff, C. (2012). Exploring the role of commercial stakeholders in open source software evolution. *In Open Source Systems : Long-Term Sustainability*, numéro 378 de IFIP Advances in Information and Communication Technology, pages 178–200. Springer Berlin Heidelberg. Cité pages 5 et 88.

Cataldo, M., Mockus, A., Roberts, J. A. et Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864–878. Cité page 18.

Chidamber, S. R. et Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493. Cité page 3.

Cohen, J., Cohen, P., West, S. G. et Aiken, L. S. (2002). *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences, 3rd Edition.* Routledge, third edition édition. Cité page 39.

Crowston, K., Wei, K., Howison, J. et Wiggins, A. (2008). Free/Libre open-source software development : What we know and what we do not know. *ACM Comput. Surv.*, 44(2): 7 :1–7 :35. Cité pages 5 et 88.

Dabbish, L., Farzan, R., Kraut, R. et Postmes, T. (2012). Fresh faces in the crowd : Turnover, identity, and commitment in online groups. *In Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, page 245–248. ACM. Cité page 23.

Dess, G. G. et Shaw, J. D. (2001). Voluntary turnover, social capital, and organizational performance. *Academy of Management Review*, 26(3):446–456. Cité page 65.

Di Penta, M. et German, D. M. (2009). Who are source code contributors and how do they change ? *In 16th Working Conference on Reverse Engineering, 2009. WCRE 2009*, pages 11–20. Cité page 15.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297. Cité page 71.

Dormann, C. F. (2011). How to be a specialist ? quantifying specialisation in pollination networks. *Network Biology*, 1(1):1–20. Cité page 52.

Duc, A. N., Cruzes, D. S., Ayala, C. et Conradi, R. (2011). Impact of stakeholder type and collaboration on issue resolution time in OSS projects. *In Open Source Systems : Grounding Research*, numéro 365 de IFIP Advances in Information and Communication Technology, pages 1–16. Springer Berlin Heidelberg. Cité page 88.

Dyba, T. et Dingsøyr, T. (2008). Empirical studies of agile software development : A systematic review. *Information and software technology*, 50(9):833–859. Cité page 2.

Efron, B. (1979). Bootstrap methods : another look at the jackknife. *The Annals of Statistics*, page 1–26. Cité page 40.

Efron, B. (1987). Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185. Cité page 41.

Fagerholm, F., Guinea, A. S., Münch, J. et Borenstein, J. (2014). The role of mentoring and project characteristics for onboarding in open source software projects. *In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, page 55 :1–55 :10. ACM. Cité page 2.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M. et others (2014). Fine-grained and accurate source code differencing. *Proceedings of the International Conference on Automated Software Engineering*. Cité pages 17 et 89.

Feldman, B. (2005). Relative importance and value. *Available at SSRN 2255827*. Cité page 58.

Fischer, M., Pinzger, M. et Gall, H. (2003). Populating a release history database from version control and bug tracking systems. *In Proceedings of the International Conference on Software Maintenance*, ICSM '03, page 23–. IEEE Computer Society. Cité page 12.

Fluri, B., Würsch, M., Pinzger, M. et Gall, H. (2007). Change distilling : Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743. Cité page 17.

Foucault, M., Falleri, J.-R. et Blanc, X. (2014). Code ownership in open-source software. *In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, page 39 :1–39 :9. ACM. Cité page 8.

Foucault, M., Palyart, M., Blanc, X., Murphy, G. et Falleri, J.-R. (2015a). Impact of developer turnover on quality in open-source software. *In Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Cité page 10.

Foucault, M., Teyton, C., Lo, D., Blanc, X. et Falleri, J.-R. (2015b). On the usefulness of ownership metrics in open-source software projects. *Information and Software Technology*, 64:102–112. Cité page 8.

Girba, T., Ducasse, S. et Lanza, M. (2004). Yesterday's weather : Guiding early reverse engineering efforts by summarizing the evolution of changes. *In Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, page 40–49. IEEE Computer Society. Cité page 3.

Girba, T., Kuhn, A., Seeberger, M. et Ducasse, S. (2005). How developers drive software evolution. *In Eighth International Workshop on Principles of Software Evolution*, pages 113–122. Cité pages 16 et 20.

Goeminne, M. et Mens, T. (2013). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986. Cité pages 4, 6, 16 et 34.

Graves, T. L., Karr, A. F., Marron, J. S. et Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661. Cité page 17.

Greevy, O., Girba, T. et Ducasse, S. (2007). How developers develop features. *In 11th European Conference on Software Maintenance and Reengineering, 2007. CSMR '07*, pages 265–274. Cité page 20.

Greiler, M., Herzig, K. et Czerwonka, J. (2015). Code ownership and software quality : A replication study. *In Proceedings of the 2015 Working Conference on Mining Software Repositories*. Cité page 20.

Groemping, U. (2006). Relative importance for linear regression in r : The package relaimpo. *Journal of Statistical Software*, 17(1):1–27. Cité page 42.

Hall, T., Beecham, S., Bowes, D., Gray, D. et Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304. Cité pages 12 et 85.

Hall, T., Beecham, S., Verner, J. et Wilson, D. (2008). The impact of staff turnover on software projects : The importance of understanding what makes software practitioners tick. *In Proceedings of the 2008 ACM SIGMIS CPR Conference on Computer Personnel Doctoral Consortium and Research*, SIGMIS CPR '08, page 30–39. ACM. Cité page 23.

Hamermesh, D. S., Hassink, W. H. J. et Ours, J. C. v. (1996). Job turnover and labor turnover : A taxonomy of employment dynamics. Open Access publications from Tilburg University 12-86873, Tilburg University. Cité page 22.

Hancock, J. I., Allen, D. G., Bosco, F. A., McDaniel, K. R. et Pierce, C. A. (2013). Meta-analytic review of employee turnover as a predictor of firm performance. *Journal of Management*, 39(3):573–603. Cité page 22.

Hassan, A. E. (2008). The road ahead for mining software repositories. *In Frontiers of Software Maintenance, 2008. FoSM 2008.*, page 48–57. IEEE. Cité page 4.

Hattori, L. et Lanza, M. (2009). Mining the history of synchronous changes to refine code ownership. *In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, page 141–150. IEEE Computer Society. Cité pages 16 et 20.

Hattori, L. P., Lanza, M. et Robbes, R. (2012). Refining code ownership with synchronous changes. *Empirical Software Engineering*, 17(4-5):467–499. Cité pages 16 et 20.

Hausknecht, J. P. et Trevor, C. O. (2011). Collective turnover at the group, unit, and organizational levels : Evidence, issues, and implications. *Journal of Management*, 37(1):352–388. Cité page 22.

Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R. et Godfrey, M. W. (2013). The MSR cookbook : Mining a decade of research. *In Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 343–352. IEEE Press. Cité pages 4, 26 et 107.

Herzig, K., Just, S. et Zeller, A. (2013). It's not a bug, it's a feature : How misclassification impacts bug prediction. *In Proceedings of the 2013 International Conference on Software Engineering*, page 392–401. Cité pages 6, 13, 34 et 46.

Hindle, A., German, D. M. et Holt, R. (2008). What do large commits tell us ? : a taxonomical study of large commits. *In Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, page 99–108. ACM. Cité page 89.

Huselid, M. A. (1995). The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of Management Journal*, 38(3):635–672. Cité page 64.

Illes-Seifert, T. et Paech, B. (2008). Exploring the relationship of history characteristics and defect count : an empirical study. *In Proceedings of the 2008 workshop on Defects in large software systems*, page 11–15. Cité page 18.

Illes-Seifert, T. et Paech, B. (2010). Exploring the relationship of a file's history and its fault-proneness : An empirical method and its application to open source programs. *Information and Software Technology*, 52(5):539–558. Cité page 18.

ISO (2011). IEC 25010 :2011 : Ingénierie des systèmes et du logiciel – exigences de qualité et évaluation des systèmes et du logiciel (SQuaRE) – modèles de qualité du système et du logiciel. *International Organization for Standardization*. Cité page 2.

Izquierdo-Cortazar, D. (2008). Relationship between orphaning and productivity in evolution and GIMP. Cité page 23.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R. et Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing ? *In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 654–665. ACM. Cité page 13.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. et Damian, D. (2014). The promises and perils of mining GitHub. *In Proceedings of the 11th Working Conference on Mining Software Repositories*, page 92–101. ACM. Cité pages 15 et 45.

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Cité page 2.

Kanter, R. M. (1976). The impact of hierarchical structures on the work behavior of women and men. *Social Problems*, 23(4):415–430. Cité pages 9 et 22.

Kawrykow, D. et Robillard, M. P. (2011). Non-essential changes in version histories. *In Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 351–360. ACM. Cité pages 13, 17 et 89.

Kim, S., Zimmermann, T., Pan, K. et Whitehead, E. J. (2006). Automatic identification of bug-introducing changes. *In 21st IEEE/ACM International Conference on Automated Software Engineering*, page 81–90. Cité page 12.

Kitchenham, B. et Pfleeger, S. L. (1996). Software quality : The elusive target. *IEEE software*, (1):12–21. Cité page 2.

Kitchenham, B. A., Dyba, T. et Jorgensen, M. (2004). Evidence-based software engineering. *In Proceedings of the 26th international conference on software engineering*, page 273–281. IEEE Computer Society. Cité pages 3 et 6.

Kochhar, P. S., Tian, Y. et Lo, D. (2014). Potential biases in bug localization : Do they matter ? *In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 803–814. ACM. Cité page 13.

Kouters, E., Vasilescu, B., Serebrenik, A. et van den Brand, M. G. (2012). Who's who in gnome : Using LSA to merge software repository identities. *In Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, page 592–595. IEEE. Cité page 16.

Kuhn, A., Ducasse, S. et Gîrba, T. (2007). Semantic clustering : Identifying topics in source code. *Information and Software Technology*, 49(3):230–243. Cité page 46.

Littlewood, B. (1978). How to measure software reliability, and how not to. *In Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, page 37–45. IEEE Press. Cité page 12.

Śliwerski, J., Zimmermann, T. et Zeller, A. (2005). When do changes induce fixes ? *In Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, page 1–5. ACM. Cité pages 12 et 13.

Louridas, P., Spinellis, D. et Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2 :1–2 :26. Cité page 39.

Meneely, A. et Williams, L. (2009). Secure open source collaboration : an empirical study of linus' law. *In Proceedings of the 16th ACM conference on Computer and communications security*, page 453–462. Cité page 19.

Meng, X., Miller, B. P., Williams, W. R. et Bernat, A. R. (2013). Mining software repositories for accurate authorship. *In 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, page 250–259. Cité page 16.

Mens, T., Claes, M., Grosjean, P. et Serebrenik, A. (2014). Studying evolving software ecosystems based on ecological models. *In Evolving Software Systems*, pages 297–326. Springer Berlin Heidelberg. Cité page 23.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R. et Jazayeri, M. (2005). Challenges in software evolution. *In Principles of Software Evolution, Eighth International Workshop on*, pages 13 – 22. Cité page 6.

Mitchell, B. et Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208. Cité page 32.

Mockus, A. (2009). Succession : Measuring transfer of code and developer productivity. *In Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 67–77. IEEE Computer Society. Cité page 23.

Mockus, A. (2010). Organizational volatility and its effects on software defects. *In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 117–126. ACM. Cité pages 9, 23, 64, 65 et 80.

Mockus, A., Fielding, R. T. et Herbsleb, J. D. (2002). Two case studies of open source software development : Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346. Cité pages 19 et 61.

Mockus, A. et Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180. Cité page 17.

Munson, J. C. et Elbaum, S. G. (1998). Code churn : A measure for estimating the impact of code change. *In Software Maintenance, 1998. Proceedings. International Conference on*, page 24–31. Cité pages 17, 36, 49, 69 et 89.

Ostrand, T. J., Weyuker, E. J. et Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355. Cité page 14.

Panciera, K., Halfaker, A. et Terveen, L. (2009). Wikipedians are born, not made : A study of power editors on wikipedia. *In Proceedings of the ACM 2009 International Conference on Supporting Group Work*, GROUP '09, page 51–60. ACM. Cité page 22.

Pearson, K. (1895). Notes on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, pages 240–242. Cité page 39.

Pinzger, M., Nagappan, N. et Murphy, B. (2008). Can developer-module networks predict failures ? *In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, page 2–12. Cité page 19.

Posnett, D., D'Souza, R., Devanbu, P. et Filkov, V. (2013). Dual ecological measures of focus in software development. *In Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 452–461. IEEE Press. Cité pages 6, 7, 19, 48, 51, 86 et 87.

Qin, X., Salter-Townshend, M. et Cunningham, P. (2014). Exploring the relationship between membership turnover and productivity in online communities. Cité page 23.

Radjenović, D., Heričko, M., Torkar, R. et Živkovič, A. (2013). Software fault prediction metrics : A systematic literature review. *Information and Software Technology*, 55(8):1397–1418. Cité page 3.

Rahman, F. et Devanbu, P. (2011). Ownership, experience and defects : a fine-grained study of authorship. *In Proceedings of the 33rd International Conference on Software Engineering*, page 491–500. Cité pages 3, 9, 20, 65 et 80.

Rahman, F. et Devanbu, P. (2013). How, and why, process metrics are better. *In Proceedings of the 2013 International Conference on Software Engineering*, page 432–441. Cité pages 2 et 48.

Ralph, P. et Kelly, P. (2014). The dimensions of software engineering success. *In Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 24–35. ACM. Cité page 2.

Ransbotham, S. et Kane, G. C. (2011). Online communities : Explaining rises and falls from grace in wikipedia. *MIS Q.*, 35(3):613–628. Cité page 22.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3): 23–49. Cité pages 7 et 52.

Riehle, D., Riemer, P., Kolassa, C. et Schmidt, M. (2014). Paid vs. volunteer work in open source. *In 2014 47th Hawaii International Conference on System Sciences (HICSS)*, pages 3286–3295. Cité pages 5 et 88.

Robles, G., Gonzalez-Barahona, J. M. et Herraiz, I. (2009). Evolution of the core team of developers in libre software projects. *In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, page 167–170. IEEE Computer Society. Cité page 23.

Sato, S., Washizaki, H., Fukazawa, Y., Inoue, S., Ono, H., Hanai, Y. et Yamamoto, M. (2013). Effects of organizational changes on product metrics and defects. *In Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 132–139. Cité page 88.

Schröter, A., Zimmermann, T., Premraj, R. et Zeller, A. (2006). If your bug database could talk. *In Proceedings of the 5th international symposium on empirical software engineering*, volume 2, page 18–20. Cité page 18.

Sharma, P. N., Hulland, J. et Daniel, S. (2012). Examining turnover in open source software projects using logistic hierarchical linear modeling approach. *In Open Source Systems : Long-Term Sustainability*, numéro 378 de IFIP Advances in Information and Communication Technology, pages 331–337. Springer Berlin Heidelberg. Cité page 23.

Shatnawi, R. et Li, W. (2008). The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of systems and software*, 81(11):1868–1882. Cité page 14.

Spearman, C. (1904). The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, 15(1):72–101. Cité page 39.

Teyton, C., Falleri, J.-R., Morandat, F. et Blanc, X. (2013). Find your library experts. *In 20th Working Conference on Reverse Engineering 2013, 14th-17th October 2013, Koblenz, Germany*, page 202–211. IEEE. Cité page 15.

Thompson, J. D. (1967). Organizations in action : Social science bases of administrative theory. SSRN Scholarly Paper ID 1496215, Social Science Research Network. Cité pages 22 et 66.

Tian, Y., Lawall, J. et Lo, D. (2012). Identifying linux bug fixing patches. *In Software Engineering (ICSE), 2012 34th International Conference on*, page 386–396. Cité page 13.

Ton, Z. et Huckman, R. S. (2008). Managing the impact of employee turnover on performance : The role of process conformance. Cité page 22.

Weyuker, E. J., Ostrand, T. J. et Bell, R. M. (2007). Using developer information as a factor for fault prediction. *In Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 8. Cité page 18.

Weyuker, E. J., Ostrand, T. J. et Bell, R. M. (2008). Do too many cooks spoil the broth ? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559. Cité pages 3, 15 et 18.

Wu, R., Zhang, H., Kim, S. et Cheung, S.-C. (2011). ReLink : recovering links between bugs and changes. *In SIGSOFT FSE*, page 15–25. Cité page 12.

Table des figures

1.1 Étapes du procédé de fouilles de données dans les dépôts logiciels [Hemmati *et al.*, 2013]. . . . 4
1.2 Répartition des contributions dans deux modules logiciels fictifs. . . . 7
1.3 Catégories de développeurs impliqués dans la rotation des effectifs. . . . 9

3.1 Étapes du procédé de fouilles de données dans les dépôts logiciels [Hemmati *et al.*, 2013]. . . . 26
3.2 Légende des diagrammes BPMN. . . . 26
3.3 Procédé d'extraction des métriques. . . . 27
3.4 Procédé d'analyse des métriques. . . . 28
3.5 Architecture de branches recherchée dans les gestionnaires de versions des projets sélectionnés. . . . 31
3.6 Exemple de *commit* de fusion. . . . 37
3.7 Exemple des résultats de corrélation : corrélation entre le nombre de lignes de code d'un module et le nombre de *commits* correctifs de celui-ci. . . . 41
3.8 Première étape de la méthode *bootstrap* 42

4.1 Répartition des contributions dans deux modules logiciels fictifs. . . . 50
4.2 Corrélation entre les métriques de répartition des contributions et **le nombre** de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*. . . . 55
4.3 Corrélation entre les métriques de répartition des contributions et **la densité** de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*. . . . 57

4.4 Importance relative des métriques dans un modèle multivarié avec pour variable dépendante le nombre de *commits* correctifs. 60

5.1 Schéma illustrant la rotation des effectifs dans un projet fictif contenant deux modules. 67
5.2 Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique. 72
5.3 Évolution du *turnover*. La ligne continue bleue (en haut) représente le nombre total de développeurs, la ligne continue violette (en bas) le nombre de développeurs persistants, la ligne pointillée verte le nombre de nouveaux arrivants, et la ligne à tirets rouge le nombre de développeurs sortants. 75
5.4 Évolution du *turnover*. Cette figure présente les ratios de nouveaux arrivants, développeurs sortants et développeurs persistants. La ligne continue violette représente le ratio de développeurs persistants, la ligne pointillée verte le ratio de nouveaux arrivants, et la ligne rouge le ratio de développeurs sortants. 75
5.5 Visualisation de l'activité des acteurs du *turnover*. Chaque rangée de la matrice correspond à un module. Plus la couleur d'une cellule est foncée, plus l'ensemble de développeurs correspondants est actif sur ce module. 76
5.6 Corrélation entre les métriques de rotation des effectifs et la densité de *commits* correctifs. Les barres d'erreurs correspondent aux intervalles de confiance à 90% et 95% obtenus avec la méthode *bootstrap*. 79
5.7 Importance relative des métriques dans un modèle multivarié avec pour variable dépendante le nombre de *commits* correctifs. 82

A.1 Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (1 sur 3). 91
A.2 Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (2 sur 3). 92
A.3 Distributions des coefficients de similarité entre les ensembles de développeurs obtenus avec des visibilités complètes et limitées de l'historique (3 sur 3). 93

Liste des tableaux

3.1 Extrait des données d'activité des développeurs produites par notre méthodologie. 38
3.2 Calcul des rangs des valeurs de métriques pour la corrélation de Spearman. $\rho = 0,825$ 40
3.3 Table de Cohen pour l'interprétation du coefficient de corrélation. 40
3.4 Projets sélectionnés dans notre jeu de données 44

4.1 Versions délimitant les périodes considérées des projets. 56

5.1 Taux de conversion des nouveaux arrivants en développeurs persistants. 74

Listings

3.1 Exemple d'une trace de *commits* incluant une opération de renommage de fichiers. Les deux nombres avant le nom du fichier représentent le nombre de lignes de code ajoutées et respectivement supprimées par chaque *commit*. 37

Abstract

Reliability of a software, *i.e.* its capacity to produce the expected behaviour, is essential to the success of software projects. To ensure such reliability, developers need to reduce the amount of bugs in the source code of the software. One of the techniques available to help developers in this task is the use of software metrics, and especially metrics related to the development process.

The general objective of this thesis is to contribute to the validation of process metrics, by studying their relationship with software reliability. These metrics, once validated, can be used in bug prediction models with the goal to guide maintenance efforts or can be used to create development guidelines. Given the extent of this domain, we chose to focus on one particular aspect of the development process, which is developers organisation, and we studied this organisation in open-source software projects.

In parallel to the validation of process metrics, we contributed to the improvement of the methodology used to extract and analyse metrics, thanks to information available in software repositories.

Keywords: *Software evolution, process metrics, quantitative studies*

Résumé

La fiabilité du logiciel, c'est-à-dire sa capacité à produire les fonctionnalités attendues, est essentielle au succès des projets de développement logiciel. Afin de garantir cette fiabilité, les développeurs ont pour objectif de réduire le nombre de bogues présents dans le code source du logiciel. Une des techniques ayant pour but d'aider les développeurs dans cette tâche est l'utilisation de métriques logicielles, et notamment de métriques liées au procédé de développement.

L'objectif général de cette thèse est de contribuer à la validation de métriques de procédé en étudiant leur relation avec la fiabilité. Ces métriques, une fois validées, pourront être utilisées dans des modèles de prédiction de bogues ayant pour but de mieux orienter les efforts de maintenance des développeurs ou pourront permettre de mettre en place des lignes de conduite relatives au procédé de développement. Devant l'étendue de ce domaine, nous avons centré nos contributions sur un aspect du procédé de développement qui est l'organisation des développeurs et avons observé cette organisation dans des projets open-source.

En parallèle de la validation de ces métriques, nous avons contribué à l'amélioration de la méthodologie permettant l'extraction et l'analyse de métriques, grâce aux informations contenues dans les dépôts logiciels.

Mots clefs : *Évolution logicielle, métriques de procédé, étude quantitative*

LaBRI — 351, Avenue de la Libération— 33400 Talence — France