université de BORDEAUX

L'impact du renommage sur les métriques de procédés

Pierre Chanson

Mémoire de stage de Master2
Encadrants: Jean-Rémy Falleri et Matthieu Foucault

LaBRI, UMR 5800
F-33400, Talence, France

Email: pierre.chanson@etu.u-bordeaux.fr,
{falleri,mfoucault}@labri.fr

25 mai 2014

Table des matières

1 Introduction

L'apparition des premiers dépôts logiciels en libre accès dans les années 90 a rendu possibles de nombreux travaux de recherche sur l'évolution logicielle. Plus particulièrement avec les dépôts de code source gérés par des outils de contrôle de versions (Version Control System, VCS) tels que SVN (2000), Mercurial (2005) ou encore Git (2005) qui contiennent l'historique de construction d'un logiciel.

C'est principalement dans le domaine de la maintenance et évolution logicielle, qui permet de comprendre les choix des développeurs lors de la création d'un logiciel, qu'il existe des études se basant sur l'analyse de ces historiques. Elles entrent dans le cadre des études "MSR" (Minning Software Repository).

La prédiction de bugs, un des défis connus du Génie Logiciel dont le but est de prédire le nombre de bugs et leurs localisations dans la version d'un logiciel avant son déploiement, utilise des informations contenues dans l'historique d'un projet. Un grand nombre d'étude sur ce sujet tentent de trouver les meilleures métriques, sur les propriétés techniques ou fonctionnelles d'un logiciel, qui serviront de prédicateurs de bugs [9].

Nagappan et al. dans l'article [29] montrent que ni les métriques tels que le nombre de ligne de code (LOC, Line Of Code) ni les métriques orientées objets tels que l'arbre d'héritage (DIT) ne peuvent être utilisés dans tous les logiciels. Depuis, beaucoup d'études plus récentes montrent que les métriques de procédés (software change metrics) donnent de bien meilleurs résultats [26, 39, 4, 10].

Les métriques de procédés se concentrent sur l'évolution d'un logiciel et mesurent les modifications subies par les entités d'un code source durant leur cycle de vie. L'hypothèse principale est que la manière dont les entités du code ont changé a un impact majeur sur leur qualité et donc sur les bugs qu'elles peuvent contenir. Il est donc primordial que les valeurs des métriques de procédés représentent au plus proche la réalité des changements.

Dans l'article [34], Radjenovic et al.identifient les trois métriques de procédés les plus utilisés. Le nombre de développeurs [39] (Number of Developers, NoD), le nombre de modifications [14] (Number of Changes, NoC) et le Code Churn [24] (CC). NoD compte le nombre de développeurs qui ont contribués à une entité. NoC compte le nombre de changements qu'a subi une entité. CC compte le nombre de lignes de code qui ont été ajoutées ou supprimées à une entité.

Calculer les métriques de procédés paraît simple à première vue. Pour un logiciel donné, cela consiste à observer tous les changements subis par chaque entité qu'il contient. Dans ce but, l'utilisation d'un gestionnaire de version est indispensable car il permet de suivre les changements effectués par tous les

développeurs sur toutes les entités.

Or au cours de son histoire, une entité du code source tel qu'un fichier, peut être renommée et/ou déplacée dans un autre dossier du projet. Ces actions sont peu ou pas prises en compte par les VCS ce qui rend le calcul des métriques plus délicat et sujet aux erreurs.

Théoriquement, si le renommage d'une entité à un moment donné de son histoire n'est pas pris en compte, le calcul d'une métrique de procédé sur ce fichier sera faussé. En effet, dans le cas où un fichier est identifié par son nom, les informations disponibles avant le renommage seront perdues. Par ailleurs, il est de notoriété commune que les refactorings, modifications architecturales qui permettent d'améliorer le code source (dont le renommage de fichiers) sont très utilisés au cours de la construction des logiciels. En pratique, nous ne connaissons pas la quantité de renommage ni son impact sur les métriques de procédés.

L'objet de nos travaux est donc d'étudier le phénomène du renommage et son impact sur les métriques de procédés. Nous présentons une étude empirique approfondie sur cinq projets open-source connus et matures avec l'intention de donner une idée du montant d'entités renommés dans les logiciels. De plus, nous étudions l'impact du renommage d'entités en calculant les trois métriques de procédés les plus utilisés sur les projets de notre corpus avec et sans considération du renommage d'entités. Nos résultats indiquent que la quantité de renommage qui apparait dans les projets peut être importante. Nous avons observé jusqu'à 99% d'entités renommés dans un projet. Nous avons aussi observé qu'il peut affecter les valeurs des métriques de procédés de manière significative, et par conséquent qu'il peut être une menace sérieuse d'études utilisant ces métriques. Basé sur nos observations, nous proposons une brève analyse de l'impact possible du renommage sur les études antérieures sur la prédiction de bugs. Nous proposons enfin de simples lignes de conduite qui aideront les chercheurs et développeurs à mieux calculer les métriques de procédés.

Pour résumer les contributions que nous apportons :
- Une étude empirique du renommage d'entités sur cinq projets open-source populaires.
- Des informations détaillées sur la quantité de renommage.
- Une analyse de l'impact du renommage sur les métriques de procédés.
- Une analyse des études antérieures qui utilisent les métriques de procédés pour la prédiction de bugs, afin d'évaluer si elles peuvent être biaisés par le renommage.
- Quelques lignes de conduite essentielles afin de limiter les risques lors des calculs des métriques de procédés.

Les résultats de nos expérimentations nous ont amenés à proposer un article

pour la conférence internationale ICSME 2014.

2 Métriques de procédés

Nous décrivons dans cette partie comment les métriques de procédés sont calculées. Puis, nous expliquons comment les fichiers renommés peuvent avoir un impact sur ces métriques. Enfin, nous décrivons comment les VCS, actuellement, traitent le renommage.

2.1 Calcul des métriques de procédés

Les métriques de procédés (change metrics) permettent de calculer à quel point une entité de code source a été modifiée au cours d'une période donnée dans l'histoire d'un logiciel. Pour la prédiction de bugs, elles sont utilisées usuellement dans la dernière période, c'est à dire entre l'avant dernière et la dernière version du projet. L'objectif est de prédire les bugs qui apparaîtront lors de la prochaine version, en particulier si cette version est une `release`, c'est à dire une sortie de logiciel dite stable. Elles ne considèrent donc que les entités étant toujours présentes à la fin de la période et qui ont étés actives dans la période. Elles excluent donc aussi les entités supprimées au cours de la période.

Un gestionnaire de versions (VCS) offre plusieurs moyens de calculer les métriques de procédés car il stocke les informations sur les entités modifiées à chaque nouvelle version, l'auteur de ces modifications, la date, etc. De plus, il permet la récupération du contenu de chaque entité et de l'ensemble d'un projet à une version donnée qu'on appelle un "`snapshot`". Pour calculer ces métriques, il est donc possible d'analyser chaque entité modifiée lors d'une période puis de garder uniquement les entités toujours présentes à la dernière version de notre période.

Plus précisément, voici comment un VCS peut être utilisé pour calculer les trois métriques de procédés les plus utilisés : le nombre de développeurs (Number of Developers, NoD), le nombre de modifications (Number of Changes, NoC) et le Code Churn (CC).

1. Nous récupérons d'abord la dernière version du projet pour obtenir les entités existantes à la fin de la période considérée. On note A cet ensemble d'entités.
2. Toujours grâce aux commandes git, nous récupérons toutes les modifications effectuées durant la période. On note C l'ensemble des modifications dans l'ordre chronologique.
3. Troisièmement, nous parcourons cet ensemble de modifications en commençant par la plus ancienne ($c_0 \in C$) jusqu'à la plus récente ($c_n \in C$)

dans le but de calculer les métriques de procédés pour chaque entité.

On note μ_a^M la valeur de la métrique M pour l'entité a. Ensuite nous calculons les métriques comme suivant (on note c_i la modification courante lors du parcours) :

NoD (nombre de développeurs) Pour chaque entité a pointé par c_i qui appartient aussi à A ($a \in A$), on ajoute à μ_a^{NoD} le nombre d'auteurs qui ont effectué les modifications c_i et qui ont modifiés a pour la première fois dans la période.

NoC (nombre de modifications) Pour chaque entité a pointé par c_i qui appartient aussi à A ($a \in A$), on ajoute 1 à μ_a^C tels que c_i indique qu'une nouvelle modification a été effectuée.

CC (Code Churn) Pour chaque entité a pointé par c_i qui appartient aussi à A ($a \in A$), on vérifie d'abord que la modification n'est pas une création d'entité. Si c'est le cas celà signifie que l'entité a été créée durant la période, donc on initialise son μ_a^{CC} à son nombre de lignes. Ensuite au prochain c_j qui cible a dans la période avec ($i < j$), on compare les deux versions et on ajoute à μ_a^{CC} le nombre de lignes ajoutées ou supprimées.

2.2 Métriques de procédés et renommage

Un VCS est donc particulièrement utile dans cet exercice. Cependant, il faut noter qu'un VCS identifie une entité par son chemin + son nom. Le chemin représente l'ensemble des dossiers parents depuis la racine du projet. On en déduit qu'un renommage du fichier ou d'un dossier, aura un impact sur le calcul des métriques. Pour expliquer cet impact, il est présenté un exemple d'historique d'un logiciel figure 2. Ce projet ne contient qu'une entité, Test.php, qui est renommé en Hello.php dans la dernière version. Dans cet exemple, nous calculons les trois métriques NoD, NoC et CC entre la version 1 et 3.

La dernière version de la période ne contient qu'une entité, `Hello.php`. En conséquence uniquement cette entité sera considérée par les techniques qui visent à prédire les bugs. Si on ne prend pas en compte le renommage, l'entité n'apparaît que dans la version 3. Ainsi, le calcul est trivial, étant donné qu'il n'y a qu'un seul développeur, alors $\mu^{NoD} = 1$. Il n'y a qu'une seule modification, la creation du fichier `Hello.php`, donc $\mu^{NoC} = 1$. La création du fichier implique l'ajout de deux lignes de codes donc $\mu^{CC} = 2$.

Par ailleurs, en prenant en compte le fait que ce fichier a été renommé, il y a trois versions à considérer qui ciblent notre entité. Le premier nom du fichier était Test.php. Les valeurs des métriques de procédés changent donc completement. Ce fichier a eu un premier auteur lors de la version 1 puis un deuxième à la version 2. Le fichier est ensuite renommé en Hello.php par un troisième auteur donc $\mu^{NoD} = 3$. Le fichier a subi trois modifications : la création du fichier

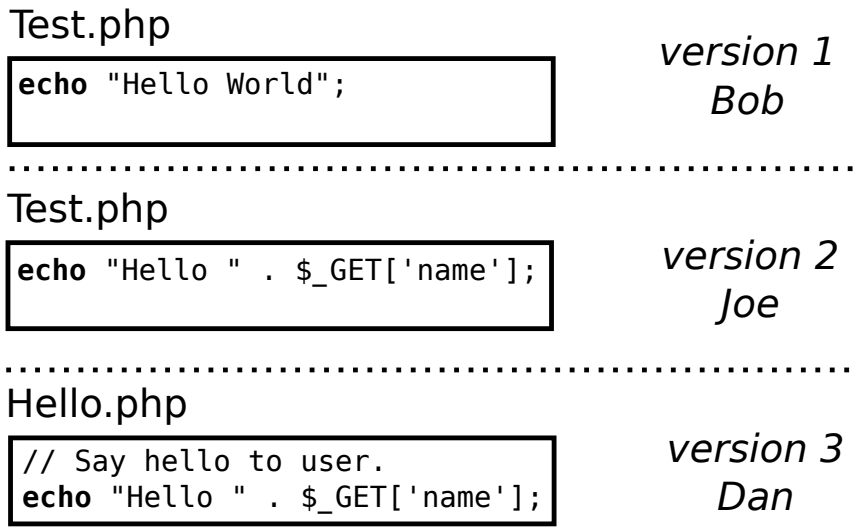

FIGURE 1 – Exemple d'un historique de projet. Le projet est composé d'un seul fichier `Test.php` qui est renommé en `Hello.php` dans la dernière version.

version 1, une modification du contenu version 2 et un renommage version 3 donc $\mu^{NoC} = 3$. Enfin, pour le Code Churn, la création du fichier implique l'ajout d'1 ligne de code, la modification de cette ligne version 2 implique la suppression d'1 ligne plus l'ajout d'1 ligne de code et la version 3, l'ajout d'1 ligne de commentaire. Nous avons donc maintenant $\mu^{CC} = 4$.

Notre exemple montre que le renommage d'entité de code source peut biaiser le calcul des métriques de procédés. Plus généralement, lorsque le renommage est pris en compte, les valeurs des métriques NoD et NoC ne peuvent qu'avoir une valeur plus grandes. En effet, étant donné que plus une entité est ancienne plus elle a de chances d'avoir un nombre de développeurs et un nombre de modifications élevés. La valeur du Code Churn quant à elle, peut augmenter ou diminuer si le renomage est pris en compte. En effet, comme nous l'avons vue dans l'exemple précédent, un renommage d'entité est considéré comme la suppression d'une entité et l'ajout d'une nouvelle entité si le renommage n'est pas pris en compte. Le renommage d'une entité dans une période induit donc que l'entité renommée a été créée durant la période. Cela entraine l'ajout du nombre de lignes de la nouvelle entité à la métrique CC, à la version où elle a été renommée. Ainsi, si l'entité est considérable, son CC sera bien plus important qu'il ne devrait. Un autre effet intéressant à remarquer est que plus une entité est renommée proche de la fin de la période, pire sera l'effet. En particulier, si elle est renommée juste avant la dernière version, alors tous les changements préalables qu'elle a subie seront perdus.

2.3 Renommage et VCS

Nous avons effectué une analyse approfondie des principaux VCS (CVS, Subversion, Git and Mercurial) dans le but de décrire les mécanismes qu'ils proposent pour traiter le renommage. Tous ces VCS sont à une granularité d'entité au niveau fichier. Comme dis précédemment, les entités, ici des fichiers donc, sont identifiés par leur chemin absolu, c'est à dire le chemin depuis la racine du projet + leur nom. Pour tout ces VCS, une modification dans le chemin d'un fichier, qui peut être due à un changement d'emplacement dans les dossiers ou à un changement de son nom, est considéré comme une suppression de fichier et création de fichier. Certains VCS proposent en complément un mécanisme pour traiter les renommages. Ce mécanisme peut être manuel ou automatique. Un mécanisme de traitement de renommage est dit manuel lorsque le développeur doit utiliser une commande particulière pour indiquer que la modification effectuée sur le fichier est un renommage. Un mécanisme est automatique lorsque le VCS propose un algorithme qui peut automatiquement détecter le renommage. De plus, ce mécanisme automatique peut être appliqué par défaut par le VCS ou de manière optionnelle lorsque le développeur doit explicitement ajouter une option de commande dans sa recherche dans l'historique.

La Table 1 résume notre étude. Alors que CVS ne gère pas du tout le renommage, SVN ou Mercural propose un mécanisme manuel de détection de renommage de fichiers. Git quant à lui propose un algorithme de détection de renommage automatique mais optionnel. Aucun de ces VCS ne propose un mécanisme automatique par défaut. Le traitement du renommage de ces VCS ne parvient cependant pas à limiter l'impact sur le calcul des métriques de procédés.

	Renaming handling		
		Automatic	
Tool	Manual	Standard	Optional
CVS			
Subversion	×		
Mercurial	×		×
Git			×

Table 1 – Traitement du renommage des principaux VCS.

Pour les VCS qui utilisent une détection manuelle, cela implique que c'est aux développeurs d'utiliser les commandes appropriées. Cependant, certaines études montrent que les développeurs n'utilisent pas ces commandes systématiquement. Le renommage peut être effectué jusqu'à 89% du temps sans utiliser

les commandes adaptées [18, 36]. De plus l'étude de Kim et al [17] montre que 51% des développeurs n'utilisent pas les commandes prévues par le VCS pour le refactoring (incluant le renommage). Ces trois études effectuées sur des projets open-source et industriels, montrent qu'il est dangereux de compter sur le fait que les développeurs utilisent les commandes adéquates pour le refactoring.

Par ailleurs, le traitement automatique optionnel du renommage nécessite une certaine connaissance des paramètres interne du VCS, afin de choisir la bonne option pour gérer le renommage. Cependant, nous n'avons jamais trouvé d'explications à propos de la configuration d'un VCS et de ses options, dans aucune étude ayant pour but la prédiction de bug avec l'utilisation de métriques de procédés. Ce n'est pas réellement surprenan. Car comme nous le décrivons dans la Section 5, dans notre analyse des études passées, aucune d'elle n'a jamais utilisée Git comme gestionnaire de version, le seul à proposer un traitement automatique optionnel du renommage.

Le mécanisme proposé par Git utilise un algorithme nommé "Origin Analysis".

2.4 "Origin Analysis"

Nous expliquons ici succinctement l'algorithme utilisé par Git pour la détection de renommage de fichiers. Celui-ci est connu sous le nom de "Origin Analysis" et est expliqué par Godfrey et al dans les articles [37, 11, 12].

Tout d'abord, il faut considérer deux versions successives d'un projet. Une version est composé d'un ensemble d'entité (fichiers, fonctions..). D'une version à la suivante, certaines entités peuvent être modifiées, certaines supprimées et d'autres ajoutées.

La première analyse proposée par Godrey pour détecter les renommages d'une version à la suivante est une analyse de Bertillonage. Cette analyse consiste à comparer entre elles les entités composantes de chaque couple "entité supprimée, entité créée", d'une version à la suivante. Cette comparaison s'effectue grâce à un choix de plusieurs métriques (LOC, cyclomatic complexity etc..). La distance Euclidienne calculée, combinée avec une technique de comparaison des noms des entités, nous obtenons une liste des renommages potentiels ordonnée. Un seuil d'acceptabilité peut alors être définit pour juger si un couple compose un renommage d'entité.

Git met donc en application cette première "Origin Analysis" uniquement, avec un seuil d'acceptabilité établi par défaut mais qui peut être configuré par la suite.

Par la suite, les analyses suivantes expliquées par Godfrey sont des améliorations de la première analyse, mais qui ne sont efficaces qu'à un niveau de

granularité plus bas, c'est à dire une finesse d'analyse plus précise, en l'occurence au niveau des fonctions. Par exemple, l'analyse des dépendances qui traque les appels de fonctions, en comparant les fonctions appelantes et appelées. Ces analyses sont basées sur plusieurs types de seuils d'acceptabilité devant à chaque fois être définis par l'utilisateur. Plus Godrey améliorera ces analyses, en prenant en compte par la suite les splits et merges de fonctions (algorithme inefficace au niveau des fichiers), plus l'utilisateur sera sollicité.

3 Méthodologie

Nous présentons ici le déroulement de nos expérimentations qui consistent à étudier le phénomène du renommage et son impact sur le calcul des métriques de procédés. L'objectif étant d'évaluer si le renommage peut biaiser de manière significative les valeurs des métriques de procédés. Pour atteindre cet objectif, nous réalisons deux expérimentations successives.

Le but de la première expérience et de calculer la quantité de renommage durant les périodes de développement des logiciels. S'appuyant sur cette première expérience, notre deuxième expérience fournit une analyse de l'impact du renommage sur les métriques de procédés dans le pire des cas.

Mais tout d'abord, nous avons besoin d'un ensemble de logiciels afin de former un coprus sur lequel appliquer nos expérimentations.

3.1 Corpus

Nous avons sélectionné un ensemble de projets sur lesquels effectuer nos expérimentations. Des projets open-source, conséquents et connus de la communauté MSR. Ces projets qui sont présentés Table 2 sont notamment régulièrement utilisés par l'équipe de Génie Logiciel au LaBRI. Ces 5 projets forment un corpus comprenant différents langages de programmation ainsi qu'un nombre de lignes de code et un nombre de développeurs moyennement élevés à élevés par rapport à l'ensemble des projets open source utilisés couramment par la communauté. Les 5 projets sont gérés sur Git afin de profiter de son mécanisme de traitement du renommage expliqué précédemment.

Les projets de notre corpus suivent des phases distinctes durant leur cycle de vie. Habituellement une période de développement commence avant qu'une première sortie du logiciel, qu'on appelle `release`, soit accessible aux utilisateurs, puis cette release est maintenue pendant qu'une autre se prépare et ainsi de suite.

Nous avons ainsi deux phases, les phases de maintenance et les phases de développement. Nous divisons ces phases en périodes. Une période est délimité par deux releases et est composé d'un ensemble de versions successives. Chaque projet contient une période qui commence à la création du logiciel et qui se

Project	Main language	Size (LoC)	Number of developers	URL
Jenkins	Java	200851	454	`github.com/jenkinsci/jenkins`
JQuery	JavaScript	41656	223	`github.com/jquery/jquery`
PHPUnit	PHP	21799	152	`github.com/sebastianbergmann/phpunit`
Pyramid	Python	38726	205	`github.com/Pylons/pyramid`
Rails	Ruby	181002	2767	`github.com/rails/rails`

TABLE 2 – Notre corpus de projets.

termine à la première release. On appelle cette période la période initiale. Les autres périodes peuvent être divisées en deux groupes, les périodes de releases majeures et les périodes de release mineures. On distingue les releases majeures des mineures par une augmentation significative du numéro de release. Par exemple $1.9 - 2.0$ pour Jquery, $3.7 - 4.0$ pour PHPUnit ou $0.13 - 1.0$ pour Rails. Usuellement, les périodes majeures contiennent un grand nombre de modifications en comparaison des périodes mineures.

Les conventions de nommage des releases sont spécifiques à chaque projet, par exemple :
- PHPunit : `3.5.0, 3.6.0` etc.
- Pyramid : `1.0, 1.1` etc.
- Jenkins : `jenkins-1_400, jenkins-1_410` etc.
- Rails : `v2.0.0, v2.1.0` etc.

Nous avons analysé manuellement les dépôts des projets de notre corpus pour identifier leur releases mineures et majeures.

Le schéma Figure 2 permet de visualiser un dépôt Git "type", avec la branche master depuis laquelle, en remontant l'historique, on peut accéder au contenu de chaque release de la phase de développement et les branches de maintenance, dont la tête (la dernière version de la branche) n'est pas accessible depuis la branche master, qui composent la phase de maintenance.

De plus, il faut noter que nous avons choisi d'exclure tous les fichiers qui ne sont pas du code source du corpus, étant donné que les métriques de procédés sont habituellement uniquement calculées sur ces fichiers.

3.2 Première expérience

L'objectif de notre première expérience est de mieux comprendre le renommage d'entités. Nous souhaitons observer quand les renommages apparaissent et en quelle quantité. Nous avons donc analysé chaque période, comme décrites

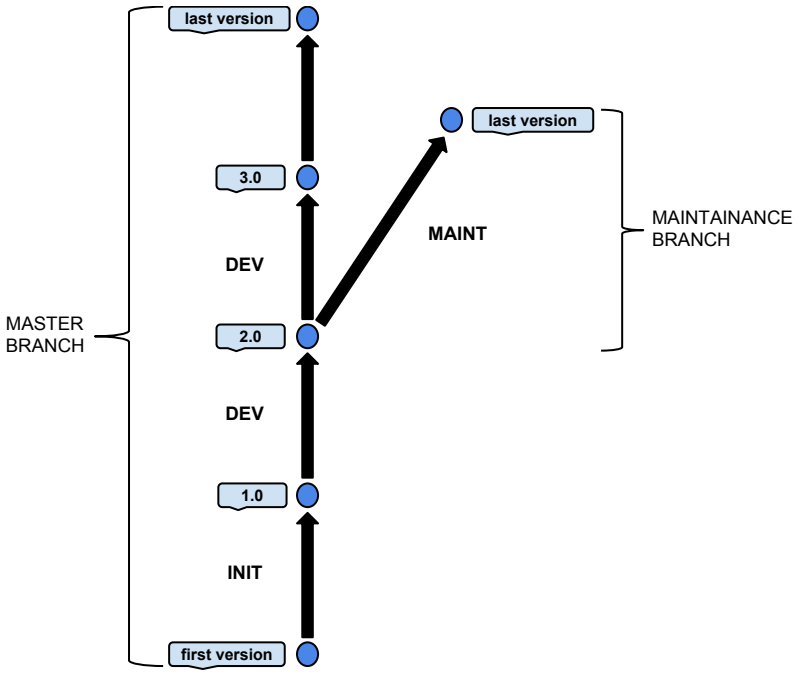

FIGURE 2 – Modèle d'architecture de dépot de code source

précédemment, sur chaque projet de notre corpus.

Pour identifier les renommages, nous comptons sur le mécanisme de Git. Voici la procédure que nous avons suivie :

1. lister les fichiers existant à la fin de la période.
2. Pour chacun de ces fichiers, extraire sa séquence de modifications durant la période en activant la détection de renommage (commande `git log -M`).
3. Calculer à partir des informations recueillies le pourcentage de fichiers $\%F_R$ qui inclue au moins un renommage dans sa séquence de modification.

$\%F_R$, calculé sur une période représente en fait le pourcentage de **l'ensemble** des fichiers du projet qui ont été renommés dans cette période en particulier. $\%F_R = \frac{\#AF_R}{\#F}$

Nous proposons en complément la même analyse que précédemment, mais en considérant uniquement les fichiers actifs dans cette même période. C'est à dire les fichiers qui ont été modifiés ou créés dans la période. On aura donc le pourcentage de fichiers **actifs** renommés dans la période : $\%AF_R = \frac{\#AF_R}{\#AF}$.
Etant donné que $\#AF <= \#F$, nous aurons toujours $\%F_R <= \%AF_R$. Cela nous permet de voir si les périodes qui contiennent le plus de renommage en interne sont aussi celles qui ont un impact sur l'ensemble du projet.

À notre connaissance, il n'existe pas d'évaluation empirique de l'algorithme utilisé par Git pour détecter les renommages. Néanmoins, nous procédons à une évaluation manuelle de son comportement dans la Section 4 et nous n'avons pas noté de faux positif sur 100 renommages aléatoires récupérés par notre analyse.

3.3 Deuxième expérience

L'objectif de la deuxième expérience est de voir si le renommage peut biaiser significativement les valeurs des métriques de procédés décrites ci-dessus. Pour ça, nous effectuons une analyse dans le pire des cas. Nous sélectionnons une période par projet, celle qui a la plus grande valeur de fichiers renommés en excluant la période initiale qui n'est généralement pas observée dans les études. Nous calculons ensuite les trois métriques avec et sans le renommage de fichiers pris en compte. Puis nous calculons la corrélation de coefficient de Spearman entre les métriques avec et sans la détection de renommage. Un coefficient élevé, proche de 1, indiquera que les métriques avec et sans détection de renommage sont très similaires alors qu'un coefficient plus petit, 0.5 et moins, indiquera que les métriques avec et sans détection de renommage sont très différentes.

4 Résultats

4.1 Première expérience

Les résultats de la première expérience sont montrés dans la Figure 3. Tout d'abord, le nombre de renommage varie beaucoup entre les projets. Par exemple, Jenkins a au plus 10% de ses fichiers renommés dans la pire période alors que PHPUnit a deux périodes à plus de 50%. Le nombre de renommages varie aussi en fonction des périodes, par exemple dans PHPUnit la période $3.6 - 3.7$ a moins de 5% de fichiers renommés alors que la période $3.7 - 4.0$ a presque 99%. En général, il y a beaucoup de périodes avec 0% de fichiers renommés.

Par rapport à la localisation de ces renommages, la période initiale semble la plus prolifique au renommage. En général, elle contient le plus grand nombre de fichiers renommés (sauf pour PHPUnit). Les périodes de développement sont plus susceptibles d'avoir des renommages que les périodes de maintenance. Ainsi, les 5 projets sont quasiment à 0% de fichiers renommés dans les périodes de maintenance. Finalement, certaines périodes de développement peuvent contenir beaucoup de renommages. Les résultats montrent que les releases majeures sont souvent les pires périodes de développement en nombre de fichiers renommés : C'est le cas pour PHPUnit et Rails alors que Jenkins

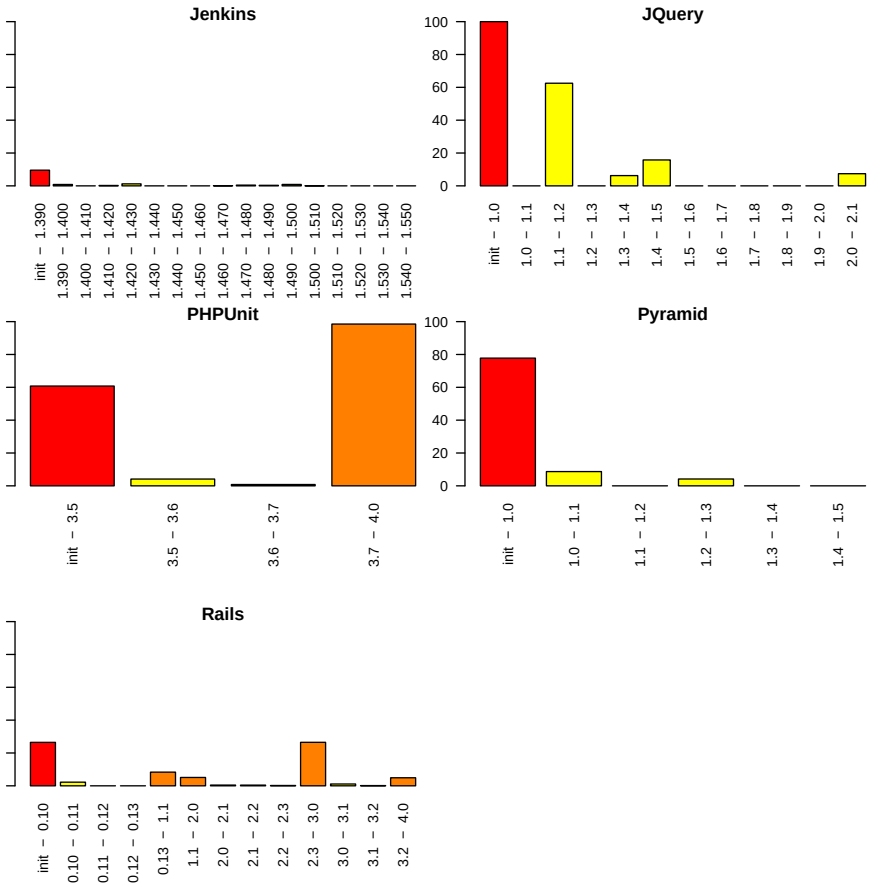

FIGURE 3 – Pourcentage de fichiers renommés ($\%F_R$) dans chaque période de chaque projet de notre corpus. La période initiale est en gris foncé, les périodes majeures en gris et les périodes mineures en gris clair.

et Pyramid ne contiennent pas de releases majeures.

En ce qui concerne le pourcentage de fichiers **actif** renommés ($\%AF_R$) dans les périodes de nos projets, la Figure 4 présente nos résultats. Nous pouvons remarquer que le taux de fichiers renommés ne change pas ou augmente légèrement pour Jenkins par exemple. Etant donné que $\%AF_R >= \%F_R$ comme nous l'avons expliqué plus tôt, on en déduit qu'une grande partie des périodes, en tout cas les périodes qui ont un impact sur l'ensemble du projet, accèdent à la totalité ou en grande partie (Jenkins) des fichiers du projet durant la période.

4.2 Deuxième expérience

Les résultats de notre deuxième expérience sont montrés dans la Table 3. Ils montrent que la corrélation de coefficients de Spearman entre les métriques de procédés avec et sans détection de renommages dépendent beaucoup de

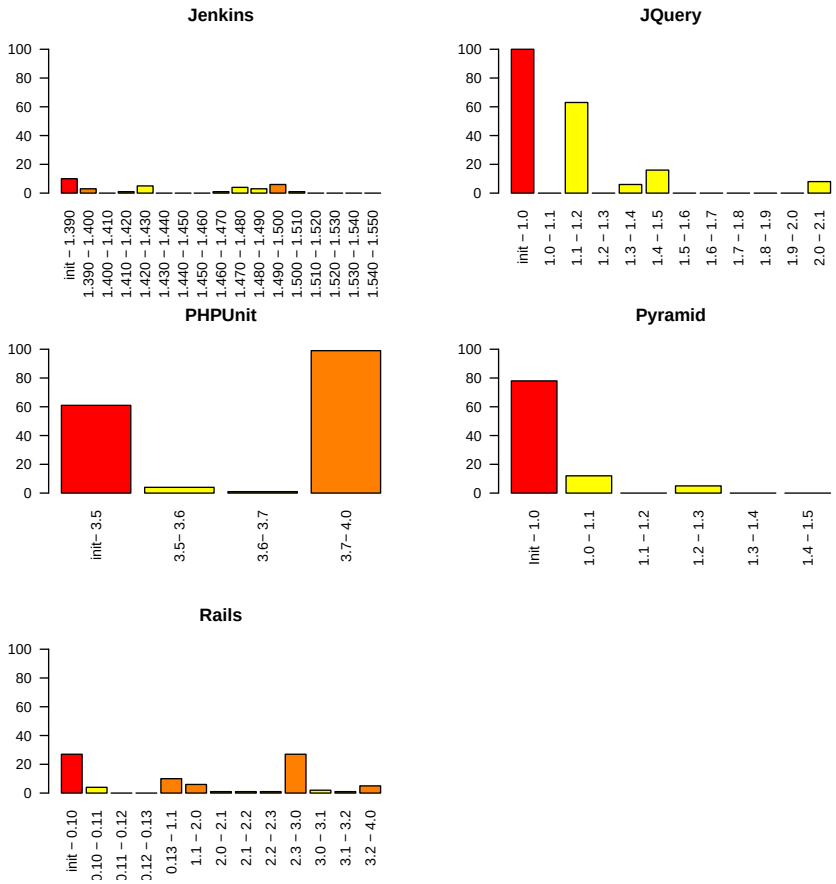

FIGURE 4 – Pourcentage de fichiers actifs renommés ($\%AF_R$) dans chaque période de chaque projet de notre corpus. La période initiale est en gris foncé, les périodes majeures en gris et les périodes mineures en gris clair.

la période et de la métrique choisie. Les métriques de procédés ne sont pas affectées par le renommage dans les projets Jenkins, Rails et Pyramid. Ainsi, le coefficient de corrélation est proche de 1 dans tous les cas. D'un autre côté, pour PHPUnit et JQuery les métriques peuvent être sévèrement impactées par le renommage. Pour JQuery, la métrique Code Churn n'est pas affectée par le renommage, mais NoD et NoC sont quant à eux significativement impactés. Pour PHPUnit, toutes les métriques sont affectées par le renommage. Sur ces deux derniers projets, la métrique la plus sensible aux renommages de fichiers est le nombre de développeurs (NoD). Sur ces deux derniers projets, la métrique la plus sensible aux renommages de fichiers est le nombre de développeurs (NoD).

Finalement, on peut noter que seules les périodes ayant eu un grand pourcentage de fichiers renommés ($\%F_R$) ont étés impactés. On peut aussi noter que des métriques biaisées par le renommage ont été calculées dans des releases majeures et mineures.

Nous avons étudié manuellement les deux périodes qui ont affecté les valeurs des métriques de procédés (JQuery 1.1 - 1.2 et PHPUnit 3.7 - 4.0). Dans ces deux périodes, la structure générale du projet a été modifié. Des renommages de dossiers de base ont étés effectués. Par conséquent, un grand nombre de fichiers a été renommé de manière transitive. C'est une pratique courante dans le développement logiciel, donc le phénomène pourrait apparaitre dans n'importe quelle période ou projet. Il est intéressant de noter que dans ces deux périodes, les changements de structure ont été effectués en grande partie dans un seul commit proche de la fin de la période.

		Change metrics		
Period	$\%F_R$	CC	NoD	NoC

TABLE 3 – La corrélation de coefficients de Spearman entre les valeurs des métriques de procédés avec et sans détection de renommage. Les codes de signification sont : *** ≤ 0.01, ** ≤ 0.05, * ≤ 0.1 et ! > 0.1. Les coefficients moyen et faible sont affichés en gras.(TODO : gérer les tableaux)

4.3 Validations et limitations

Notre étude fait l'hypothèse que les renommages détectés par Git sont corrects. Néanmoins, nous n'avons pas rencontré d'évaluation empirique de l'algorithme de détection de renommage de Git et par conséquent nous n'avons pas confiance dans ses résultats. Afin d'atténuer cette menace, nous avons tiré au hasard 100 renommages détectés par Git lors de notre expérimentation. Nous avons évalué manuellement chaque renommage de fichier pour vérifier si la détection était correcte. Vérifier que la détection soit correcte consiste à s'assurer que le contenu du fichier avant et après renommage soit très similaire, et qu'aucun autre fichier ajouté dans la même version n'ait un contenu similaire. Cette analyse manuelle nous a révélé que 100% des renommages étaient correctement détectés. Même si nous sommes conscients que l'algorithme Git peut donner des faux positifs, cette expérience montre que l'utilisation de Git dans la détection de renommage reste raisonnable. Nous n'avons pas analysé les faux négatifs, c'est à dire les vrais renommages de fichier non détectés par Git, car ça ne risquerait pas de fausser nos conclusions. Cela ne pourrait que diminuer les coefficients de corrélation si nous l'avons sous-estimé.

En raison d'un très petit nombre de fichiers, les valeurs des coéfficient de corrélation pour JQuery sont basses pour NoD et NoC. Cependant, la signification de la corrélation n'est pas une menace en ce qui concerne nos conclusions.

Nous avons uniquement évalué l'effet du renommage sur trois métriques de procédés. L'effet du renommage pourrait être différent (mieux ou pire) pour les autres métriques de procédés. Notre solution pour limiter ce risque était d'analyser les métriques de procédés les plus utilisées comme il est montré dans [34]. Les autres métriques de procédés sont souvent basées sur ces trois métriques comme la métrique *code ownership* [4] ou *module activity focus* [33].

Pour la métrique NoD, nous n'avons pas appliqué un algorithme de merge d'identités [13]. Il pourrait en résulter des valeurs incorrectes. Cependant, ce phénomène est susceptible d'arriver pour les deux calculs de métrique avec et sans le renommage, donc le risque qu'il invalide nos conclusions est faible.

Concernant notre conclusion sur la quantité de renommage, le corpus utilisé ne garantit pas qu'elle puisse être généralisée. En effet, nous avons uniquement utilisé des projets open-source, alors que les projets industriels sont connus pour être sensiblement différents. En ce qui concerne la validité sur les projets open-source, notre corpus est trop petit pour généraliser cette conclusion.

5 Analyse des études antérieures

Dans cette partie, nous procédons à l'analyse d'études antérieures sur la prédiction de bugs qui ont utilisé les métriques de procédés pour la prédiction de bugs. Nous évaluons si les valeurs des métriques de procédés pourraient être biaisées en regardant la façon dont ils ont recueilli leurs données.
Enfin, nous donnons quelques lignes de conduite à suivre pour aider les chercheurs et développeurs à éviter l'impact que le renommage d'entités pourrait avoir sur les métriques de procédés.

5.1 Analyse des études antérieures

Premièrement, comme nous l'avons montré dans la Section **??**, il est important de remarquer que les périodes contenant un taux de fichiers renommés élevés sont rares. Ainsi, la majeure partie des études antérieures ne devraient pas être affectées par ce phénomène. De plus, même dans le cas d'analyses sur des périodes contenant un taux élevé de fichiers renommés, les résultats de ces analyses pourraient également être améliorés, car les métriques de procédés auraient probablement été sous-estimées. Toutefois, plusieurs études antérieures peuvent être affectées par le renommage, comme nous le signalerons dans la suite de cette section. Quantifier de tels effets sur les études antérieures est hors de portée de notre sujet, mais nous fournissons tout de même certaines lignes de conduite à respecter pour les études futures dans Section 5.2.

Dans notre examen des études antérieures, nous n'analysons que 26 des articles référencés dans [34]. Les articles qui utilisent les trois métriques de procédés, CC, NoD ou NoC. Cependant, certaines des autres études référencées dans cet article utilisent d'autres métriques de procédés et pourraientt donc aussi être affectées par le renommage.

15 de ces études analysent des projets industriels, [2, 14, 16, 19, 24, 26, 30, 27, 28, 25, 31, 32, 39, 38, 41]. Aucune de ces études ne parle de renommage. Mais le manque d'informations récoltées sur les VCS utilisés et sur le projet en lui-même, ne nous permet pas de savoir si le renommage pouvait avoir un impact sur ces projets. Néanmoins, l'article de Kim et al [17] explique que les développeurs dans son étude effectuent des opérations de refactoring, dont du renommage, sans utiliser les outils du VCS appropriés. Ainsi, ces études pourraient être impactées par le renommage en fonction des outils utilisés et des habitudes de développement.

11 études analysent des logiciels open-source [6, 3, 5, 8, 8, 7, 15, 20, 21, 22, 23, 35]. Les VCS utilisés dans ces études sont CVS ou Subversion. CVS ne gère pas le renommage et Subversion uniquement de manière manuelle ce qui est dangereux comme expliqué dans l'article [18, 36]. Seulement deux de ces études [22, 23] parlent de renommage, dans leur set de données ou dans les "Threats to validiy". Pour réduire le risque d'erreur dans leurs expérimentations, ces deux études ont supprimé systématiquement tous les fichiers ajoutés ou supprimés durant les périodes analysées. C'est un bon moyen d'éviter de calculer des métriques de procédés biaisés, mais cela implique aussi de supprimer inutilement du jeu de données un nombre significatif de fichiers.

5.2 Lignes de conduite

Les résultats de nos deux expérimentations nous permettent de déduire de simples lignes de conduite pour calculer les métriques de procédés. Voici donc nos recommandations :

- Eéviter de calculer ces métriques durant les périodes intitiales. En effet, ces périodes contiennent habituellement une quantité de renommage significative. Comme nous l'avons vu, les deux périodes majeures et mineures peuvent contenir un beaucoup de renommage, bien que les releases majeures semblent plus sujettes au renommage.
- Utiliser systématiquement un algorithme de détection de renommage, afin d'éviter d'analyser les mauvaises périodes. Git propose un algorithme dédié qui semble avoir une bonne précision, mais un rappel inconnu. Par conséquent, l'utilisation de projets gérés avec Git parrait la méthode la plus simple pour diminuer les risques du renommage. Des algorithmes de détection plus avancés sont décrits dans la littérature [1, 18, 36]. Ils ont

étés validés par des études empiriques donc ils pourraient réaliser une analyse meilleure que celle de Git.
- Pour les métriques de procédés calculées à des niveaux de granularité plus fin que celui des fichiers, utiliser les algorithmes d"Origin Analysis" tels que [40]. Ces algorithmes sont efficaces au niveau de granularité des fonctions.
- Le renommage d'entités peut être un risque important, penser à indiquer systématiquement comment il a été traité dans les études futures.

6 Conclusion

Dans ce rapport, nous avons évalué l'impact du renommage d'entités sur les valeurs des métriques de procédés logiciels. Nous avons effectué une étude empirique sur cinq projets open-source connus et matures. Nous avons observé que les périodes initiales des projets sont plus enclines à contenir du renommage que les autres périodes. Plus important, nous avons constaté que d'autres périodes peuvent contenir une quantité importante de renommage, en particulier celles correspondantes à la mise au point de releases majeures. Enfin, nous avons observé que le renommage pouvait biaiser considérablement les valeurs des métriques de procédés.

Par conséquent, nous avons mis en évidence que les chercheurs et développeurs devraient être prudents lors du calcul des métriques de procédés pour éviter de biaiser leurs conclusions. Nous avons pu leur faire quelques recommandations, comme d'éviter le calcul des métriques de procédés lors des périodes initiales et d'utiliser un algorithme de détection de renommage lors du calcul des métriques de procédés sur les autres périodes.

Suite à cette étude, il est prévu d'évaluer la précision des algorithmes existants de détection de renommage et d'évaluer l'impact de la fusion de code (code merging) sur les métriques de procédés.

Références

[1] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*, pages 31–40, September 2004.

[2] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1) :2 – 17, 2010. SI : Top Scholars.

[3] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone ? In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, page 59–73, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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, New York, NY, USA, 2011. ACM.

[5] Bora Caglayan, Ayse Bener, and Stefan Koch. Merits of using repository metrics in defect prediction for open source projects. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, page 31–36, Washington, DC, USA, 2009. IEEE Computer Society.

[6] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 135–144, October 2009.

[7] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41, May 2010.

[8] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches : a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5) :531–577, 2012.

[9] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5) :675–689, September 1999.

[10] E. Giger, M. Pinzger, and H.C. Gall. Can we predict types of code changes ? an empirical analysis. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 217 –226, June 2012.

[11] Michael Godfrey and Qiang Tu. Tracking structural evolution using origin analysis. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, page 117–119, New York, NY, USA, 2002. ACM.

[12] M.W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2) :166–181, 2005.

[13] Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8) :971–986, August 2013.

[14] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7) :653–661, July 2000.

[15] Timea Illes-Seifert and Barbara Paech. 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, May 2010.

[16] T.M. Khoshgoftaar, R. Shan, and E.B. Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 301–310, 2000.

[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, page 50 :1–50 :11, New York, NY, USA, 2012. ACM.

[18] T. Lavoie, F. Khomh, E. Merlo, and Ying Zou. Inferring repository file structure modifications using nearest-neighbor clone detection. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 325–334, October 2012.

[19] Lucas Layman, Gunnar Kudrjavets, and Nachiappan Nagappan. Iterative identification of fault-prone binaries using in-process metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, page 206–212, New York, NY, USA, 2008. ACM.

[20] Paul Luo Li, Mary Shaw, and Jim Herbsleb. Finding predictors of field defects for open source software systems in commonly available data sources : A case study of openbsd. In *IN : METRICS '05 : PROCEEDINGS OF THE 11TH IEEE INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, IEEE COMPUTER SOCIETY*, page 32, 2005.

[21] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 18, 2010.

[22] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical*

Software Engineering and Measurement, ESEM '08, page 309–311, New York, NY, USA, 2008. ACM.

[23] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ACM/IEEE 30th International Conference on Software Engineering*, page 181–190, 2008.

[24] John C. Munson and Sebastian G. Elbaum. Code churn : A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings. International Conference on*, page 24–31, 1998.

[25] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318, November 2010.

[26] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, page 284–292, New York, NY, USA, 2005. ACM.

[27] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures : An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, page 364–373, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Nachiappan Nagappan, Thomas Ball, and Brendan Murphy. Using historical in-process and product metrics for early estimation of software failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, page 62–74, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, page 452–461, New York, NY, USA, 2006. ACM.

[30] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality : an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, page 521–530, 2008.

[31] Allen P. Nikora and John C. Munson. Building high-quality software fault predictors. *Software : Practice and Experience*, 36(9) :949–969, 2006.

[32] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, page 19 :1–19 :10, New York, NY, USA, 2010. ACM.

[33] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 452–461, Piscataway, NJ, USA, 2013. IEEE Press.

[34] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics : A systematic literature review. *Information and Software Technology*, 55(8) :1397–1418, August 2013.

[35] Adrian Schröter, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. If your bug database could talk. In *Proceedings of the 5th international symposium on empirical software engineering*, volume 2, page 18–20, 2006.

[36] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Incremental origin analysis of source code files. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[37] Qiang Tu and M.W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension, 2002. Proceedings*, pages 127–136, 2002.

[38] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Using developer information as a factor for fault prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 8, 2007.

[39] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Do too many cooks spoil the broth ? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5) :539–559, October 2008.

[40] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA : a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 325–334, New York, NY, USA, 2010. ACM.

[41] X. Yuan, T.M. Khoshgoftaar, E.B. Allen, and K. Ganesan. An application of fuzzy clustering to software quality prediction. In *Application-Specific Systems and Software Engineering Technology, 2000. Proceedings. 3rd IEEE Symposium on*, pages 85–90, 2000.