



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

23 mai 2014

Table des matières

1	Introduction	3
2	Métriques de procédés	4
2.1	Calcul des métriques de procédés	4
2.2	Métriques de procédés et renommage	6
2.3	Renommage et VCS	7
2.4	“Origin Analysis”	9
3	Méthodologie	9
3.1	Corpus	9
3.2	Première expérience	11
3.3	Deuxième expérience	11
4	Résultats	12
4.1	Première expérience	12
4.2	Deuxième expérience	13
4.3	Validations et limitations	13
5	Analyse des études passés	15
5.1	Analysis of Past Studies	15
5.2	Guidelines	16
6	Conclusion	16

1 Introduction

L'apparition des premiers dépôts logiciels en libre accès dans les années 90 a rendu possible 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" (Mining Software Repository).

La prédiction de bugs, un des défis connus du Génie Logiciel dont le but est de prédire le nombre de bugs et leur localisations dans la version d'un logiciel avant son déploiement, utilise des informations contenues dans l'historique d'un projet. Un grand nombre d'études 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 tout les logiciels. Depuis, beaucoup d'études plus récentes montrent que les métriques de procédés (software change metrics) donne 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 étant 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 de développeurs qui ont contribué à une entité. NoC compte le nombre de changements qu'a subie 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 subies par chaque entité qu'il contient. Dans ce but, l'utilisation d'un gestionnaire de version est indispensable tels qu'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 tels qu'un fichier,

peut être renommée et/ou déplacée dans un autre dossier du projet. Ces actions sont peut 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 disposons pas de chiffres pour en connaître l'ampleur.

(TODO : changer la fin) L'objet de nos travaux est donc d'étudier les pistes qui peuvent nous conduire à mettre en évidence les renommages, les récupérer et effectuer certaines statistiques. Dans un premier temps, nous présentons état de l'art sur les méthodes utilisées pour détecter le refactoring, les logiciels qui ont été étudiés, les métriques de procédés ainsi que les VCS.

Compte tenu de l'état de l'art nous exposons la problématique à résoudre, qui nous a conduit à redéfinir le renommage et les niveaux de granularités. Puis, réaliser une analyse manuelle sur un premier projet afin de récupérer les renommages réels. Ces travaux se sont poursuivis par la définition d'un modèle, le choix d'un ensemble de projets cohérent pour faire nos propres expérimentations et la création d'un outil pour récupérer les renommages.

Enfin, nous décrivons comment calculer certaines métriques de procédés et mesurons l'impact du renommage.

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 calculés. 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 été actives dans la période. Elles excluent donc aussi les entités supprimés 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 cela veut dire que l'entité a été créée durant la période, donc on initialise son μ_a^{CC} à son nombre de lignes. Ensuite au

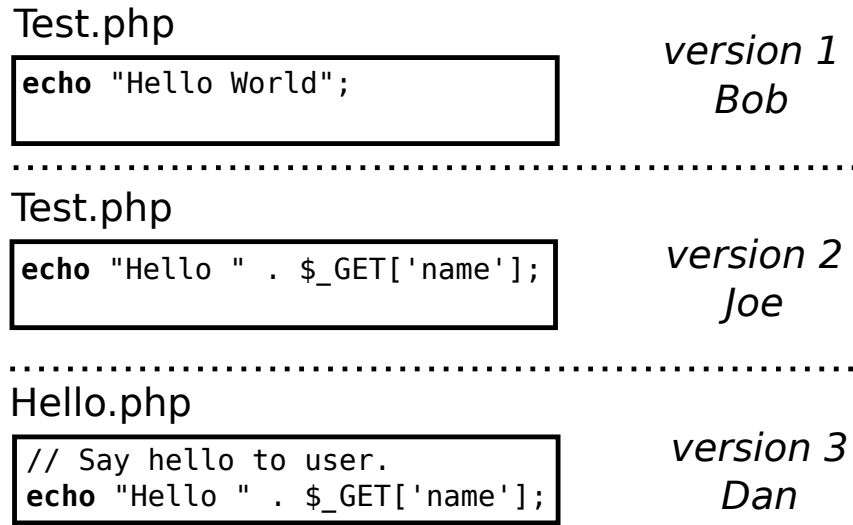


FIGURE 1 – Exemple d’un historique de projet. Le projet est composé d’un seul fichier `Test.php` qui est renommé en `Hello.php` dans la dernière version.

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 création 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 en ce qui concerne l’entité. Le premier nom du fichier était `Test.php`. Les valeurs des métrique de procédés change donc complètement. 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 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, les valeurs des métriques NoD et NoC peuvent seulement être plus grande lorsque le renommage est pris en compte étant donné que plus une entité est ancienne plus elle a de chances d'avoir un nombre de développeurs et de modifications élevés. La valeur du Code Churn quant à elle, peut augmenter ou diminuer si le renommage est pris en compte. En effet, comme nous l'avons vu dans l'exemple, 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 entraîne 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 préalable 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 absolue, 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 développeurs 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 algorithm 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 Mercurial 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.

Tool	Renaming handling		
	Manual	Automatic	
		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 surprenant comme nous décrivons dans la Section 5, dans notre analyse des études passés, qu'aucune d'elle n'a jamais utilisé 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. Deux ensembles d’entités (fichiers, fonctions..) qui composent leurs versions respectives. Certaines entités ayant été modifiées, certaines supprimées et d’autres ajoutées. La première analyse est une analyse de Bertillonage qui consiste à choisir un nombre de métriques, puis comparer les entités avec ces métriques. On compare alors les entités supprimées avec les entités ajoutées d’une version à l’autre. Grace à la distance Euclidienne calculée à partir des métriques combinés avec une comparaison des noms des entités, nous obtenons une liste des renommages potentiels.

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’occurrence au niveau des fonctions. Par exemple, l’analyse de dépendance qui traque les appels de fonctions, en comparant les fonctions appelantes et appelées. Ces analyses sont basées sur des seuils d’acceptabilité défini par l’utilisateur. Plus Godfrey 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. (TODO : détailler ?)

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 est 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 corpus sur lequel appliquer nos expérimentations.

3.1 Corpus

Nous avons sélectionné un ensemble de projets sur lesquels effectuer nos expérimentation. Des projets open-source, conséquents et connues de la communauté MSR. Ces projets qui sont présentés Table 2 sont notamment régulière-

ment 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.

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.

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 délimitée par deux releases et est composée d'un ensemble de versions successives. Tout d'abord, chaque projet contient une période qui commence à la création du logiciel et qui se 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 releases 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 analysés manuellement les dépôts des projets de notre corpus pour identifier leur releases mineures et majeures.

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 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.

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 aléatoire récupérés par notre outil.

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 2. 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.

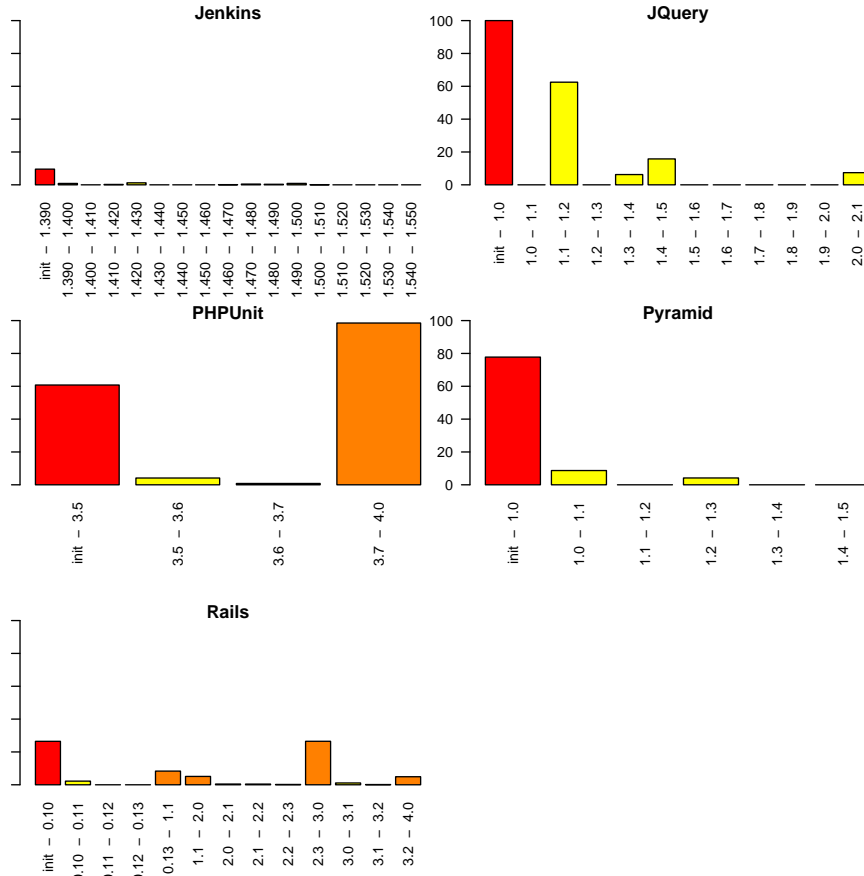


FIGURE 2 – Pourcentage de fichiers renommés ($\%F_R$) dans chaque période de chaque projet de notre corpus. La période initiale est en gris foncé, les périodes majeures en gris et les périodes mineures en gris clair.

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 et Pyramid ne contiennent pas de releases majeures.

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 dépend beaucoup de 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 tels que 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 été 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). Par conséquent un grand nombre de fichiers ont été renommés de manière transitive. C’est une pratique courante dans le développement logiciel, donc le phénomène pourrait apparaître 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.

4.3 Validations et limitations

Our study makes the assumption that renamed files detected by Git tool are correct. However, we did not encounter any empirical evaluation of the

	Period	$\%F_R$	Change metrics		
			CC	NoD	NoC
jenkins	1.420 \triangleright 1.430	0.89	1	1	1
jquery	1.1 \triangleright 1.2	0.98	0.98	0.08*	0.08*
phpunit	3.7.0 \triangleright 4.7.0	0.6	0.6	0.38	0.38
pyramid	1.0 \triangleright 1.1	0.96	0.97	1	1
rails	2.3.0 \triangleright 3.0.0	0.98	0.98	0.96	0.96

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)

Git renaming detection algorithm and therefore we have no confidence on its results. To mitigate this threat, we drew at random 100 renamed files detected by Git in our corpus. We manually assessed each detected file renaming to check if it was correctly detected. Checking if a detected file renaming is correct consists in ensuring that the file content is very similar, and that no other file added in the same version has a similar content. This manual analysis revealed that 100% of the detected renamed files were correctly detected. Even though we are aware that Git algorithm can yield false positives, this experiment shows that using Git as an oracle to detect renaming is reasonable. We did not analyzed the false negatives (true file renaming not detected by Git) as this is not a threat to our conclusion, since it will most likely lower the correlation coefficients if underestimated.

Due to a very small number of files, significance values of the correlation coefficients for the JQuery project are low for NoD and NoC. Nevertheless, significance of the correlation is not a threat with regard to our conclusion.

We only assessed the effect of renaming on three change metrics. The effect of renaming could be different (worse or better) for other change metrics. To mitigate this risk we used the most used change metrics as shown in [34]. Other change metrics are often based upon these metrics (such as *code ownership* [4] or *module activity focus* [33]).

For the NoD metric, we did not apply an identity merging algorithm [13]. This could result in incorrect values. However, this phenomenon is likely to happen for both metrics with and without renaming, therefore the risk that it invalidates our conclusion is low.

Concerning our conclusion on the amount of renaming, the corpus we used do not guarantee that it is generalizable. Indeed, we only used open-source

projects, while industrial projects are known to be significantly different. Regarding the validity among open-source projects, our corpus is too small to generalize our conclusion.

5 Analyse des études passées

In this section, we proceed to an analysis of the past studies that used change metrics to predict defects. We evaluate if the values of change metrics could be biased by analyzing how they collected their data. Finally, we give some guidelines to help researchers and practitioners to avoid the impact that artifact renaming has on change metrics.

5.1 Analysis of Past Studies

Firstly, it is important to remark that as we have shown in Section ??, periods containing a high amount of renaming are rare. Therefore, most of the past studies should be not affected by this phenomenon. Additionally, even in the case of using periods having a significant amount of renaming, the results of such studies could also be improved, because change metrics would probably have been underestimated. Nevertheless, several past studies can be impacted by renaming, as we will point out in the remainder of this section. Quantifying such effect on these past studies is out of the scope of this paper, but we provide guidelines for future studies in Section 5.2.

In this analysis of past studies, we include only the 26 studies referenced in [34] that use the CC, NoD or NoC metrics. However several other studies referenced in [34] use slightly different change metrics and could also be impacted on renaming.

Nous nous sommes donc intéressés aux études passées qui pouvaient traiter les trois métriques de procédés cités ci-dessus dans la prédiction de bugs, et vérifié si ces études avaient considéré le renommage de fichiers. L'article [34] de Rajenovi et al référence 26 études sur ce sujet.

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'information 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 validity". 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 Guidelines

According to the results of our two experiments, we deduce several simple guidelines to compute change metrics. Firstly, we recommend to avoid computing such metrics during initial periods of projects at all costs. Indeed, these periods usually contain a significant amount of renaming. As we have seen, both major and minor periods can contain a significant amount of renaming, although major release seems more prone to renaming. In any case, we recommend to systematically use a renaming detection algorithm, to avoid picking up the wrong period. Git provides a dedicated algorithm that seems to have a good precision, but an unknown recall. Therefore using projects managed by Git seems the easier way to lower the threat of renaming. More advanced renaming detection algorithms are also described in the literature : [1, 18, 36]. They have been empirically validated so they might perform better than Git's algorithm, and are the only choices if the chosen corpus contains project that are not managed by Git. Finally, for change metrics computed at finer level of granularity than files, we recommend the use of origin analysis algorithms such as [40]. These algorithms usually work at the granularity of the functions. Finally, as artifact renaming can be a significant threat, we recommend to systematically indicate how it was dealt with in future studies.

6 Conclusion

In this article, we assessed the impact of artifact renaming on the values of software change metrics. We conducted an empirical study on five popular and mature open-source software projects. We observed that initial periods of projects are more prone to renaming than the other periods. More importantly, we observed that other periods can contain a significant amount of renaming, especially those corresponding to the development of major releases. Finally, we observed that renaming can significantly skew the values of change metrics. Therefore, researchers and practitioners should be very careful when computing change metrics. We recommend to avoid computing change metrics on initial periods. Finally, we highly recommend to systematically use a renaming

detection algorithm when computing change metrics on other periods as it can change significantly the values of change metrics.

As a future work, we plan to evaluate the accuracy of existing renaming detection algorithms. We also plan to evaluate the impact of code merging on software change metrics.

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 Symposium 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.