Impact of Renaming on Software Change Metrics

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

22 mai 2014

Table des matières

1 Introduction

L'apparition des premiers dépots logiciels en libre accès dans les années 90, des stockages centralisés de donnés, 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 du "Reverse Engineering", qui permet decomprendre les choix des développeurs lors de la création d'un logiciel, qu'il existe des études qui se basent sur l'analyse de ces historiques. Elles entrent dans le cadre des études "MSR" (Minning Software Repository).

De même, la prédiction de bugs, un des défis connus du Génie Logiciel dont le but est de prédire le nombre de bugs et leur localisation dans la prochaine version d'un logiciel, utilise des informations contenues dans l'historique d'un projet. Le principe repose sur les métriques de procédés comme prédicateurs de bugs.

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 générer. Il est donc primordial que les mesures des métriques de procédés représentent au plus proche la réalité des changements.

Or au cours de son histoire, un fichier peut être renommé et/ou déplacé dans un autre dossier du projet.

Théoriquement, si le renommage d'un fichier 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 récoltées avant le renommage seront perdues. Par ailleurs, il est de notoriété commune que le refactoring, modifications architecturales qui permettent d'améliorer le code source, dont le renommage de fichiers, est très présent dans le développement des logiciels à succès d'aujourd'hui. En pratique, nous ne disposons pas de chiffres pour en connaitre l'ampleur.

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 cohérant pour faire nos propres expérimentations et la creation 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 ICSME 2014.

2 Etat de l'art

2.1 Evolution logiciel et refactoring

Pour commencer à comprendre l'évolution des logiciels et la place du refactoring, nous avons d'abord cherché les articles qui mentionnaient le refactoring. Puis nous avons cherché des études de MSR qui pourraient mettre en avant des chiffres à propos du refactoring, par exemple un pourcentage d'opérations de refactoring ou de commits contenant du refactoring. On peut régulièrement lire en introduction d'articles dans le domaine, des propos sur l'importance du refactoring, qui inclut le renommage, et sur l'intérêt des techniques de compréhension de l'évolution des architectures et structures des logiciels. Il est de notoriété commune que les logiciels à succès sont généralement amenés à évoluer dans le temps et à se restructurer après la découverte de bugs, l'ajout de fonctionnalités ou l'adaptation à l'environnement dans lequel ils évoluent. Le maintien d'un tel logiciel passe par la compréhension des choix d'architecture pris par le passé, c'est-à-dire par son histoire. [30, 7, 12].

Néanmoins nous n'avons pas obtenu de chiffres précis sur le nombre de renommage, le nombre d'opérations de refactoring ou autres. Uniquement dans l'étude de Kim et al, qui nous donne un pourcentage d'opérations de renommage sur le nombre opérations de refactoring. Ce qui ne nous donne pas l'importance du refactoring par rapport au projet entier.

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

Table 1 – Traitement du renommage des principaux VCS.

2.2 Les gestionnaires de version

Intéressons-nous aux outils disponibles pour la gestion de code source. Il existe un certain nombre de gestionnaires de versions tel que SVN, CVS, Mercurial ou Git qui pourraient être compatibles avec notre étude étant donné que nous avons simplement besoin de versions, c'est-à-dire un état du projet à un moment donné de son histoire, à comparer entre elles. Nous avons néanmoins étudié les VCS en détail et découvert que tous ne gèrent pas le renommage de fichiers de la même manière. 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 et optionnel. 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 [13, 29]. De plus l'étude de Kim et al 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.

2.3 "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 [30, 7, 8].

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'occurence 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 Godrey améliorera ces analyses, en prenant en compte par la suite les splits er merges de fonctions (algorithme inefficace au niveau des fichiers) plus l'utilisa-

teur sera sollicité. (TODO : détailler ?)

2.4 Métriques de procédés et évolution logiciel

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. Elles sont utilisés 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 particulié 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.
Radjenovic et al [27] identifient trois métriques de procédés les plus utilisés pour la prédiction de bugs : Le nombre de développeurs [32] (Number of Developers, NoD), le nombre de modifications [9] (Number of Changes, NoC) et le Code Churn [19] (CC). Nous donnerons une définition et une méthode plus précise pour les calculer dans la section de nos expériences..

2.5 Métriques et renommages

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 [27] de Rajenovi et al référence 26 études sur ce sujet.

15 de ces études analyses des projets industriels, [1, 9, 11, 14, 19, 21, 24, 22, 23, 20, 25, 26, 32, 31, 33]. 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 [12] 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 [4, 2, 3, 6, 6, 5, 10, 15, 16, 17, 18, 28]. 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 [13, 29]. Seulement deux de ces études [17, 18] 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.

3 Problématique

Nous n'avons pas trouvé d'études traitant le renommage. Ces études ont elles volontairement ou non omis le renommage ?
La problématique qui se pose est donc la suivante :
Quelle est la quantité de renommages ou déplacements de fichiers dans les projets ? Où interviennent-ils ? Ont-ils un réel impact sur les métriques de procédés ?

4 Première analyse à fine granularité

Le premier travail réalisé durant le stage a consisté à réaliser une analyse manuelle des renommages dans les VCS. Nous avons sélectionné 100 commits de manière aléatoire dans un projet et étudié le renommage d'entités dans ces commits.

Nous avons choisi Hibernate-ORM, un projet open-source connue et assez gros, 750000 LOC, avec suffisamment de développeurs, 138, et en JAVA afin de pouvoir différencier les renommages à différents niveaux de granularité.
Nous avons pris 4 niveaux de granularité : Dossier, fichier, classe et fonction.
Nous avons défini l'identité d'une entité de code source par son path plus un type.

dossier = folder/folder/ | FOLDER
fichier = folder/folder/file | FILE
classe = folder/folder/file | CLASS
fonction = folder/folder/file#func(types) | FUNC
classe interne = folder/folder/file$class | CLASS

Nous nous sommes ensuite intéressés à la localisation des renommages. Sont-ils plus proches des releases majeurs que des releases mineurs ? Nous nous sommes aussi demandé si Git détectait ces renommages au niveau des fichiers, et si cela pouvait être un indicateur pour tous les changements d'identité.

Nous avons donc décidé d'utiliser à partir de maintenant la détection de renommages de Git, afin de couvrir un grand nombre de commits dans plusieurs projets et plusieurs langages de programmation et donc nous fixer à un seul niveau de granularité, les fichiers.

5 Un modèle

Nous définissons maintenant le modèle Figure 1, que les projets que nous allons analyser devrons respecter. Ce modèle représente une architecture pour le dépôt de code source dérivé de l'architecture Git Flow.

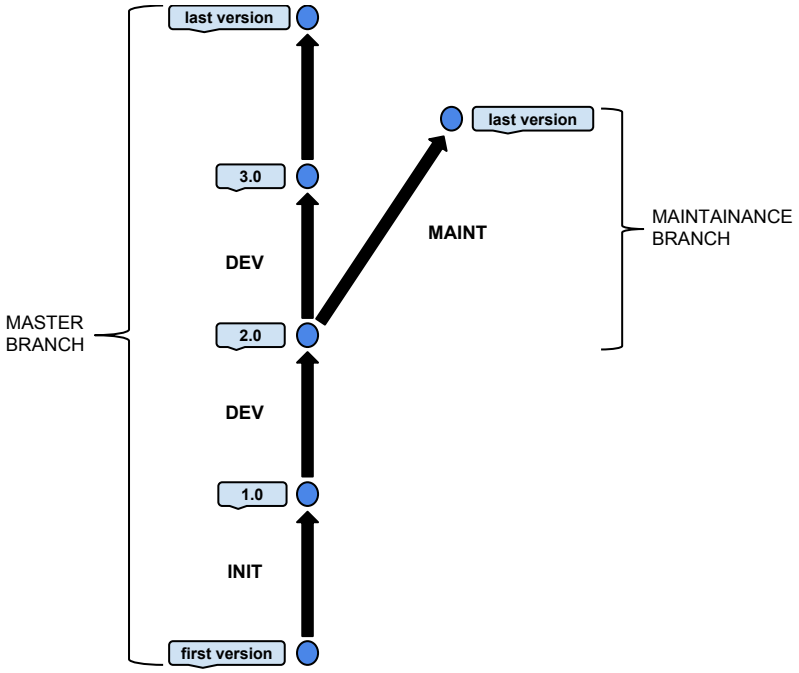

FIGURE 1 – Modèle d'architecture de dépot de code source

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

Nous avons ainsi deux types de branches, les branches de maintenance et la branche master. Nous divisons les branches en périodes, c'est-à-dire en une séquence de commits, la branche master contient une période initiale (init) entre la première version du logiciel (incluse) et la première release, puis des périodes de développement (dev) entre chaque release. Chaque projet contient des releases majeures, qui correspondent à des points-clés du projet, des périodes susceptibles de contenir beaucoup de changements, et des 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. De l'autre côté les branches de maintenance sont divisées en périodes (maint) entre chaque release.

6 Un corpus de projets

Nous avons sélectionné un ensemble de projets sur lesquels effectuer nos expérimentations qui respectent le modèle défini. Des projets open-source, conséquents et connues de la communauté MSR. Nous avons un ensemble de projets utilisés par l'équipe de Génie Logiciel au LaBRI qui respectent le modèle avec des branches de maintenances identifiées. Les 5 projets qui sont donnés Table 2 forment un corpus pour notre prochaine expérience. Il comprend différents langages de programmation ainsi qu'un nombre de lignes de code et un nombre de développeurs moyennement élevés à évélevés par rapport aux projets open source utilisés couramment par la communauté. Les 5 projets sont gérés sur Git afin de profiter de la détection automatique des renommages (section).

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.

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.

7 Analyse à gros grain

7.1 Première experience

L'objectif de notre première expérience est de mieux comprendre le renommage d'entités. En particulier, d'observer quand les renommages apparaissent et en quelle quantité. Nous avons donc analysé chaque période comme décrites précédemment sur chaque projet de notre corpus. Nous comptons donc sur la détection automatique de renommage de Git et l'utilisons dans notre outil développé en Ruby pour obtenir nos chiffres (détails dans la section résultats). (TODO pourquoi Ruby ?) Nous suivons ces trois étapes entre chaque période :

1. On liste les fichiers existant à la fin de la période.
2. Pour chacun de ces fichiers, on extrait sa séquence de modification durant la période en activant la détection de renommage (commande `git log`

`-M`)

3. On calcule à partir des informations receuillies. Par exemple le pourcentage de fichiers $\%F_R$ qui ont été renommés au moins une fois durant la période.

Plus précisément voici le déroulement général de notre outil :

Avant tout nous avons besoin de la liste des tags de release trié dans l'ordre chronologique. Cette partie ne peut être automatisé en raison des conventions de nom de tags différentes entre chaque projets. 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.

Enfin voici le déroulement du processus :

Nous parcourons les branches distantes (`remotes`) du dépôt Git du projet. Toute les branches sont considérés comme branche de maintenance, sauf la branche `master`. La plus part du temps, les branches spécifiques de maintenance contiennent "-stable" dans leur nom, mais si une branche est listée ici, cela veut dire que la tête de branche, c'est à dire sa dernière version, est séparée et n'est pas joignable de la branche `master`. C'est donc une branche de maintenance depuis son dernier `merge` avec master, c'est à dire depuis leur dernière fusion. Il est laissé à la discrétion de chacun d'éliminer les branches qui ne sont pas spécifiquement de maintenance au préalable, mais dans nos expériences, ces branches n'ont pas faussé notre étude. En général les branches présentes sur le dépôt qui ne sont ni `master` ni de maintenance ne contennaient peux ou pas de `commits`.

La branche `origin/master` contient la partie initiale (init) et la partie de développement (dev).
Le travail est donc réalisé sur :

- Les branches de maintenance
- La branche `origin/master`

Si la branche est `origin/master`, le travail est divisé encore dans les périodes :

- Du premier commit(inclue) jusqu'au premier tag de release.
- Du premier jusqu'au dernier tag (une release après l'autre forme une période).*

(* nous ne considérons pas la période du dernier tag jusqu'à la tête de la branche master, car la release n'étant pas terminé, le nombre de commit ne sera pas de la taille d'une release normale et les chiffres obtenus sur cette partie ne seront donc pas pertinants)

Ainsi, pour chacune de nos trois conditions, maintenance(maint), période initiale(init) et releases(dev) qui formes l'ensemble des périodes que nous

analysons, nous allons générer le `log`. Le `log` donné par la commande `git log` contient l'ensemble des `commits` dans nos périodes avec toute les informations nécessaires pour chacun d'eux.

La majeure partie du travail réalisé se situe sur ces `log`. Nous avons choisis de travailler sur un `log` général plutôt qu'en récupérant l'historique de chaque fichier (avec l'option ``-follow'') l'un après l'autre pour des raisons de performances. Particulièrement sur les gros projets tels que Rails ou Jenkins, cette dernière technique n'est presque pas réalisable. De plus elle n'est réalisable que sur les fichiers présent à la tête de la branche et pas à partir d'une version antérieure, ce qui impliquerais un travail supplémentaire pour configurer le dépôt à chaque release sur la branche `master`.

Nous listons les fichiers existants à la fin de chaque période. Dans notre expérience nous écartons les fichiers qui ne sont pas du code source. Enfin l'algorithme principal, qui parcours des informations contenus par les `log` dans l'ordre chronologique, va nous permettre de suivre les renommages et les modifications et ainsi reconstruire l'histoire des fichiers. Un renommage typique ressemble à ``rename bob/{henry ⇒ josef}/george.py (86%)''. Si ''bob/henry/george.py'' à été enregistré au préalable par notre algorithme nous suivrons maintenant ''bob/josef/george.py'' à la place. Il y a de nombreux cas particuliers à prendre en compte et de nombreuses applications possible par rapport aux informations récupérés ainsi. On peut choisir de ne considérer que le dernier nom d'un fichier par exemple ou stocker tout ses noms précédents. on élimine finalement tout les fichiers qui ne sont pas dans notre liste de établie au préalable.

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 partie conclusion et nous n'avons pas noté de faux positif sur 100 renommages aléatoire récupérés par notre outil.(TODO partie vérif ?)

7.2 Deuxième expérience

7.2.1 Calcul des métriques

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. 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.

Par ailleurs, il faut noter qu'un VCS identifie une entité par son chemin

FIGURE 2 – 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.

+ nom de fichier. 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 le nombre de développeurs (NoD) entre la version 1 et 3.

Le NoD d'une entité de code source au cours d'une période de son histoire correspond au nombre de développeurs ayant été identifiés comme auteurs d'une modification sur l'entité pendant la période donnée.

En ne prenant pas en compte le renommage, la dernière version ne contient qu'une entité. C'est donc cette entité uniquement qui sera considérée. De plus, l'identité exacte de cette entité n'apparaît que lors de la version 3. Le calcul des métriques est donc trivial, NoD= 1.

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. Ce fichier a 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. Le NoD est donc de 3.

Nous avons choisi de nous concentrer sur les trois métriques de procédés identifiés plus tôt pour mesurer l'impact du renommage. En utilisant des scripts Ruby pour mesurer les métriques sur nos projets, voici plus précisément notre procédure :

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 creation d'entité. Si c'est le cas celà 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 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 suprimées.

7.2.2 Calcul de corrélation

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.(TODO expliquer ?) On sélectionne la période de nos projets 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.

8 Résultats

8.1 Résultats de la 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.

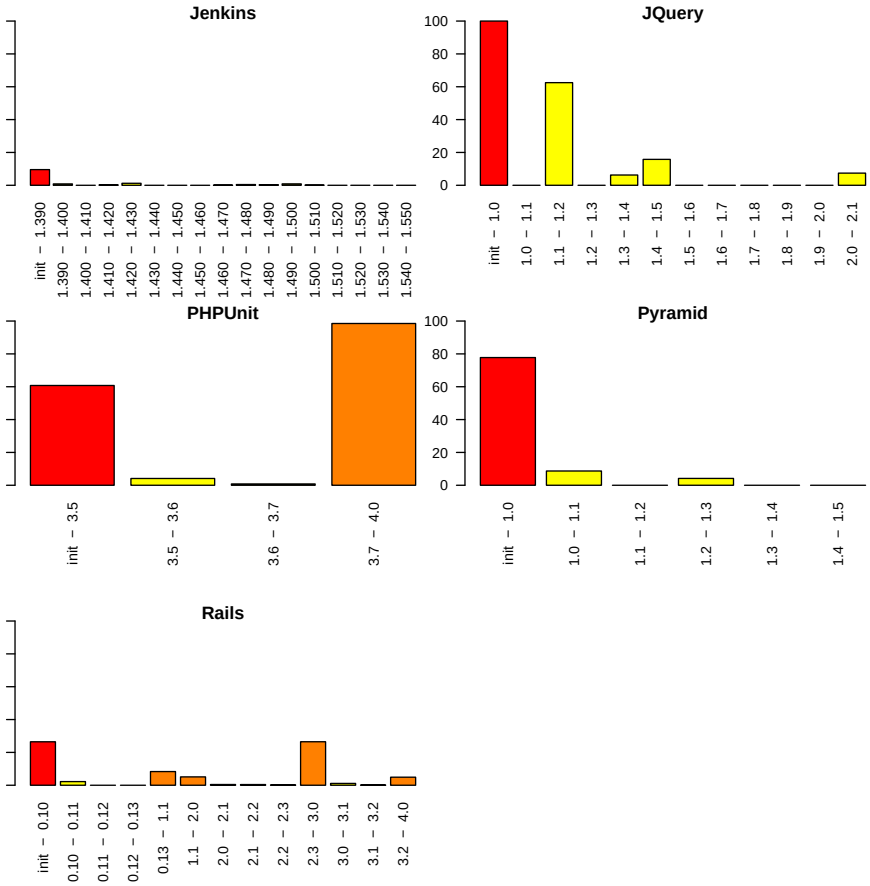

FIGURE 3 – Pourcentage de fichiers renommés ($\%F_R$) dans chaque période de chaque projet de notre corpus. La période initiale est en gris foncé, les périodes majeures en gris et les périodes mineures en gris clair.

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 (voir tableaux détails). 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.

Le détail des résultats obtenus par notre outil de détection de renommage sont montrés dans les tables en annexes Table 4, Table 5, Table 6, Table 7 et Table 8. Ces tables mettent en avant les valeurs suivantes :

- *Nombre de fichiers (#F) :* nombre de fichiers dans le projet à la dernière version de la période..
- *Nombre de fichiers actifs (#AF) :* nombre de fichiers créés, suprimés, copiés ou renommés durant la période et présents à la dernière version de la période.
- *Pourcentage de fichiers actifs (%AF) :* $\%AF = \frac{\#AF}{\#F}$.
- *Nombre de fichiers actifs renommés ($\#AF_r$) :* nombre de fichiers actifs qui ont étés renommés.
- *Pourcentage de fichiers renommés ($\%F_R$) :* $\%F_R = \frac{\#AF_R}{\#F}$.
- *Pourcentage de fichiers actifs renommés ($\%AF_R$) :* $\%AF_R = \frac{\#AF_R}{\#AF}$.

8.2 Résultats de la 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 dépendend 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 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). Par conséquent un grand nombre de fichiers ont étés renommés 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 intéressent 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
jenkins 1.420 ▷ 1.430	0.89	1	1	1
jquery 1.1 ▷ 1.2	0.98	0.98	0.08*	0.08*
phpunit 3.7.0 ▷ 4.7.0	0.6	0.6	0.38	0.38
pyramid 1.0 ▷ 1.1	0.96	0.97	1	1
rails 2.3.0 ▷ 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 coéfficiants moyen et faible sont affichés en gras.(TODO : gérer les tableaux)

8.3 vérifications

9 Conclusion

Références

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

Annexes

Period	Kind	Renaming metrics				
		$\#F$	$\#AF$	$\%AF$	$\%F_r$	$\%AF_r$
init ▷ 1.390	INIT	795	795	100.0	9.55	9.55
1.390 ▷ 1.400	DEV	825	236	28.6	0.84	2.96
1.400 ▷ 1.410	DEV	824	153	18.56	0.0	0.0
1.410 ▷ 1.420	DEV	843	354	41.99	0.23	0.56
1.420 ▷ 1.430	DEV	814	190	23.34	1.22	5.26
1.430 ▷ 1.440	DEV	818	126	15.4	0.0	0.0
1.440 ▷ 1.450	DEV	838	142	16.94	0.0	0.0
1.450 ▷ 1.460	DEV	861	140	16.26	0.0	0.0
1.460 ▷ 1.470	DEV	869	101	11.62	0.11	0.99
1.470 ▷ 1.480	DEV	886	114	12.86	0.45	3.5
1.480 ▷ 1.490	DEV	909	124	13.64	0.33	2.41
1.490 ▷ 1.500	DEV	922	130	14.09	0.86	6.15
1.500 ▷ 1.510	DEV	933	114	12.21	0.1	0.87
1.510 ▷ 1.520	DEV	948	174	18.35	0.0	0.0
1.520 ▷ 1.530	DEV	962	191	19.85	0.0	0.0
1.530 ▷ 1.540	DEV	895	129	14.41	0.0	0.0
1.540 ▷ 1.550	DEV	904	159	17.58	0.0	0.0
1.409	MAINT	823	28	3.4	0.0	0.0
1.424	MAINT	791	47	5.94	0.0	0.0
1.447	MAINT	828	57	6.88	0.0	0.0
1.466	MAINT	864	70	8.1	0.0	0.0
1.480	MAINT	898	113	12.58	0.0	0.0
1.509	MAINT	937	182	19.42	0.0	0.0
1.532	MAINT	901	182	20.19	0.0	0.0

TABLE 4 – Amount and location of renaming in Jenkins

		Renaming metrics				
Period	Kind	$\#F$	$\#AF$	$\%AF$	$\%F_r$	$\%AF_r$
init ▷ 1.0	INIT	4	4	100.0	100.0	100.0
1.0 ▷ 1.1	DEV	12	12	100.0	0.0	0.0
1.1 ▷ 1.2	DEV	8	8	100.0	62.5	62.5
1.2 ▷ 1.3	DEV	11	10	90.9	0.0	0.0
1.3 ▷ 1.4	DEV	16	16	100.0	6.25	6.25
1.4 ▷ 1.5	DEV	19	19	100.0	15.78	15.78
1.5 ▷ 1.6	DEV	21	19	90.47	0.0	0.0
1.6 ▷ 1.7	DEV	22	18	81.81	0.0	0.0
1.7 ▷ 1.8	DEV	26	26	100.0	0.0	0.0
1.8 ▷ 1.9	DEV	26	23	88.46	0.0	0.0
1.9 ▷ 2.0	DEV	29	28	96.55	0.0	0.0
2.0 ▷ 2.1	DEV	81	81	100.0	7.4	7.4
1.8	MAINT	26	3	11.53	0.0	0.0
1.9	MAINT	27	20	74.07	0.0	0.0

TABLE 5 – Amount and location of renaming in JQuery

		Renaming metrics				
Period	Kind	$\#F$	$\#AF$	$\%AF$	$\%F_r$	$\%AF_r$
init ▷ 3.5	INIT	130	130	100.0	60.76	60.76
3.5 ▷ 3.6	DEV	121	121	100.0	4.13	4.13
3.6 ▷ 3.7	DEV	124	124	100.0	0.8	0.8
3.7 ▷ 4.0	DEV	135	135	100.0	98.51	98.51
1.3	MAINT	13	13	100.0	0.0	0.0
2.3	MAINT	46	46	100.0	0.0	0.0
3.0	MAINT	206	206	100.0	0.0	0.0
3.1	MAINT	194	16	8.24	0.0	0.0
3.2	MAINT	285	170	59.64	0.0	0.0
3.3	MAINT	342	330	96.49	0.58	0.6
3.4	MAINT	395	0	0.0	0.0	0
3.5	MAINT	124	0	0.0	0.0	0
3.6	MAINT	126	0	0.0	0.0	0
3.7	MAINT	125	1	0.8	0.0	0.0
4.0	MAINT	135	1	0.74	0.0	0.0
4.1	MAINT	120	0	0.0	0.0	0

TABLE 6 – Amount and location of renaming in PHPUnit

		Renaming metrics				
Period	Kind	$\#F$	$\#AF$	$\%AF$	$\%F_r$	$\%AF_r$
init ▷ 1.0	INIT	45	45	100.0	77.77	77.77
1.0 ▷ 1.1	DEV	46	34	73.91	8.69	11.76
1.1 ▷ 1.2	DEV	58	41	70.68	0.0	0.0
1.2 ▷ 1.3	DEV	72	65	90.27	4.16	4.61
1.3 ▷ 1.4	DEV	72	52	72.22	0.0	0.0
1.4 ▷ 1.5	DEV	68	54	79.41	0.0	0.0
1.0	MAINT	45	12	26.66	0.0	0.0
1.1	MAINT	46	13	28.26	0.0	0.0
1.2	MAINT	58	15	25.86	0.0	0.0
1.3	MAINT	72	6	8.33	0.0	0.0
1.4	MAINT	72	11	15.27	0.0	0.0
1.5	MAINT	68	0	0.0	0.0	0

TABLE 7 – Amount and location of renaming in Pyramid

		Renaming metrics				
Period	Kind	$\#F$	$\#AF$	$\%AF$	$\%F_r$	$\%AF_r$
init ▷ 0.10	INIT	170	170	100.0	26.47	26.47
0.10 ▷ 0.11	DEV	180	101	56.11	2.22	3.96
0.11 ▷ 0.12	DEV	192	93	48.43	0.0	0.0
0.12 ▷ 0.13	DEV	217	123	56.68	0.0	0.0
0.13 ▷ 1.1	DEV	312	253	81.08	8.33	10.27
1.1 ▷ 2.0	DEV	373	335	89.81	5.09	5.67
2.0 ▷ 2.1	DEV	436	298	68.34	0.45	0.67
2.1 ▷ 2.2	DEV	467	284	60.81	0.42	0.7
2.2 ▷ 2.3	DEV	482	257	53.31	0.2	0.38
2.3 ▷ 3.0	DEV	570	565	99.12	26.49	26.72
3.0 ▷ 3.1	DEV	630	503	79.84	1.11	1.39
3.1 ▷ 3.2	DEV	684	451	65.93	0.14	0.22
3.2 ▷ 4.0	DEV	734	689	93.86	4.9	5.22
1.2	MAINT	361	146	40.44	0.0	0.0
2.0	MAINT	381	68	17.84	0.0	0.0
2.1	MAINT	442	118	26.69	0.22	0.84
2.2	MAINT	470	67	14.25	0.0	0.0
2.3	MAINT	512	260	50.78	0.0	0.0
3.0	MAINT	575	307	53.39	0.0	0.0
3.1	MAINT	638	266	41.69	0.15	0.37
3.2	MAINT	686	297	43.29	0.0	0.0
4.0	MAINT	734	255	34.74	0.0	0.0
4.1	MAINT	756	117	15.47	0.0	0.0

TABLE 8 – Amount and location of renaming in Rails