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

18 mai 2014

Table des matières

1 Introduction

L'accès aux dépôts logiciels a rendu possible de nombreux travaux de recherche sur l'évolution logicielle. Plus particulièrement, les dépôts de code source gérés par des outils de contrôle de versions (Version Control System, VCS, comme SVN, Mercurial ou encore Git) contiennent l'historique de construction d'un logiciel. Des études se basent sur l'analyse de ces historiques. Principalement dans le "Reverse Engineering", la compréhension des choix des développeurs lors de la création d'un logiciel, ou encore 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 prochaine version d'un logiciel. Cette étude se base sur les métriques de procédés comme prédicteurs de bugs. Les métriques de procédés se concentrent sur l'évolution d'un logiciel et mesures 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 changés a un impact majeur sur la qualité de leur prédiction de bugs.
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, si on identifie le fichier par son nom, on perdra les informations récoltés avant le renommage. Par ailleurs on peut penser que le refactoring, dont le renommage de fichiers, est très present dans le dévelopement des logiciels à succès d'aujourd'hui. En pratique, nous n'avons pas de chiffres pour le montrer.
Dans un premier temps nous effectuerons une étude de l'existant sur les méthodes utilisés pour détecter le refactoring, les logiciels qui ont étés étudiés, les métriques de procédés ainsi que les VCS. Puis nous choisirons un ensemble de projets cohérant pour faire nos propres expérimentations, nous définirons un niveau de granularité et nous ferons une analyse manuelle des projets choisis pour récupérer les renommages réels. Par la suite nous définirons un modèle et nous utiliserons un outils pour récupérer les renommages. Enfin nous définirons comment calculer certaines métriques de procédés et mesurerons l'impact du renommage. Les résultats de nos expérimentations aménerons à une publication dans la conférence ICSME 2014.

2 Etat de l'art

2.1 Evolution logiciel et refactoring

On peut régulièrement lire en introduction d'articles des propos sur l'importance du refactoring, ce qui inclue le renommage. Sur l'intérêt des techniques de compréhension de l'évolution des architectures et structures des logiciels. Les logiciels à succès sont généralement amenés à évoluer dans le temps, à se restructurer etc, après découverte de bugs, l'ajout de fonctionnalités, l'adaptation à l'environnement dans lequel ils évoluent. Le maintient d'un tel logiciel passe par la compréhension des choix d'architecture pris par le passé, par son histoire [**?**, **?**, **?**]. Néanmoins on obtient pas de chiffres précis sur le nombre de renames. Uniquement dans l'étude de Kim et al un pourcentage de rename sur les operations de refactoring.

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

Table 1 – Handling of renaming of the main VCS tools.

2.2 Les outils

Intéressons nous aux outils disponibles pour la gestion de code source. Il existe un certains nombre de gestionnaires de versions tels que SVN, CVS, Mercurial ou Git qui pourraient être compatible 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 néhanmoins étudié les VCS en détails 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 méchanisme manuel de détection de renommage de ficheirs. Git quant à lui propose un algorithme de détection de renommage automatique et optionnel. Pour les VCS utilisent une détection manuelle, celà implique que c'est aux développeurs d'utilisé les commandes appropriés. Cependant certaines études montrent que les développeurs n'utilise pas ces commandes systématiquement. Le renommage peut être effectués jusqu'à 89% du temps sans utilisés les commandes adaptés [**?**, **?**]. 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 éffectués 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 rapidement l'algorithme utilisé par Git pour la détection de renommage de fichiers. Celui-ci est connu sous sous le nom de "Origin Analysis" et est expliqué par Godfrey et al dans les articles, [**?**, **?**, **?**]
Tout d'abord il faut considérer deux versions succéssives d'un projet. Deux ensembles d'entités (fichiers, fonctions..) qui composent leur versions respectives. Certaines entités ayant étés modifiés de la version à la suivante, certaines supprimés et d'autres ajoutés. 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 supprimés avec les entités ajoutés d'une version à l'autre. Grace à la distance Euclidienne calculée à partir des métriques combiné avec une comparaison des nom des entités, nous obtenons une liste des renommages potentiels.
Les analyses suivantes expliqués par Godfrey sont des ameliorations de la première

analyse mais qui ne sont efficace qu'à un niveau de granularité plus bas, au niveau des fonctions. Par exemple l'analyse de dépendance qui tracke les appels de fonctions, en comparent les fonctions appelantes et appelés. Ces analyse sont basés sur des seuils d'acceptabilités définit par l'utilisateur. Plus Godrey améliorera ces analyses, en prenant en compte par la suite les splits er merges de fonctions (algorithme inéficace au niveau des fichiers) plus l'utilisateur sera solicité.

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 à été modifiée au cours d'une période donnée dans l'histoire d'un logiciel. On les utilises usuellement dans la dernière période avant la dernière version, l'objectif étant de prédire les bugs qui apparaitront lors de la prochaine release. Elles ne considère donc que les entités étant toujours présentes à la fin de la période et qui ont étés actives dans la période.
Radjenovic et al [?] identifient trois métriques de procédés les plus utilisés pour la prédiction de bugs : Le nombre de développeurs [?] (Number of Developers, NoD), Le nombre de modifications [?] (Number of Changes, NoC) et le Code Churn [?] (CC). Nous donnerons une définition et une méthode précise pour les calculer dans nos expérimentations.

2.5 Métriques et Renommage Existant

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

15 de ces études analyses des projets industriels, [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Aucune de ces études ne parle de renommage, mais le manque d'informations récoltés sur les VCS utilisés et sur le projet en lui même ne nous permet pas de savoir si le renommage aurait pu avoir un impact sur ces projets. Néanmoins, l'article de Kim et al [?] explique que les dévelopeurs dans son étude effectuent des opérations de refactoring, dont du renommage, sans utiliser les outils du VCS appropriés. Ainsi ces études pourraient être impactés par le renommage en fonction des outils utilisés et des habitudes de dévelopement.

11 études analyse des logiciels open-source [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. 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 [?, ?]. Seulement deux de ces études [?, ?] parlent de renommage dans leurt set de données ou dans les "Threats to validiyy". Pour réduire le risque d'erreur dans leurs expérimentations, ces dux études ont supprimés systématiquement out les fichiers ajoutés ou supprimés durant les périodes analysés. C'est un bon moyen de d'éviter de calculer des métriques de procédés biaisés, mais celà implique aussi de

supprimer inutilement du jeux de données un nombre significatif de fichiers.

3 Problématique

Nous n'avons pas réellement trouvé d'études traitant le renommage. Ces études ont-elles volontairement ou non omis le renommage ?
La problématique qui se pose est donc, **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 grain fin

Le premier travail réalisé durant le stage a été de faire une étude manuelle des renommages dans les VCS. Nous avons sélectionnés 100 commits de manière aléatoire dans un projet et étudié le renommage d'entités dans ces commits.
Nous avons choisit Hibernate-ORM, un projet open-source connue et assez gros, 750000 LOC, avec suffisament de développeurs, 138, et en JAVA afin de pouvoir differencier les renommages à different niveaux de granularités.
Nous prenons 4 niveaux de granularité : Dossier, fichier, classe et fonction. Nous définissons 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

Un renommage ou un déplacement seraient donc un changement dans l'identité. Enfin nous avons diférencié le changement d'identité direct, lorsque l'entité elle même est direcetement modifié, du changement d'identité induit, un changement d'identité de l'entité due à un changement d'identité d'un de ses parents. Si l'entité est changé de manière induite puis de manière directe, on compte uniquement le changement direct.
Nous obtenons 17% de commits contenant des changements d'identité à tout les niveaux de granularité. (TODO tableau)

5 Un modèle

Nous nous sommes ensuite intéressé à la location des renommages. Sont-ils plus proches des releases majeurs que des releases mineurs ? Nous nous sommes aussi demandé si Git détectais ces renommages au niveau des fichiers, et si celà pouvait être un indicateur pour tout les changements d'identités.

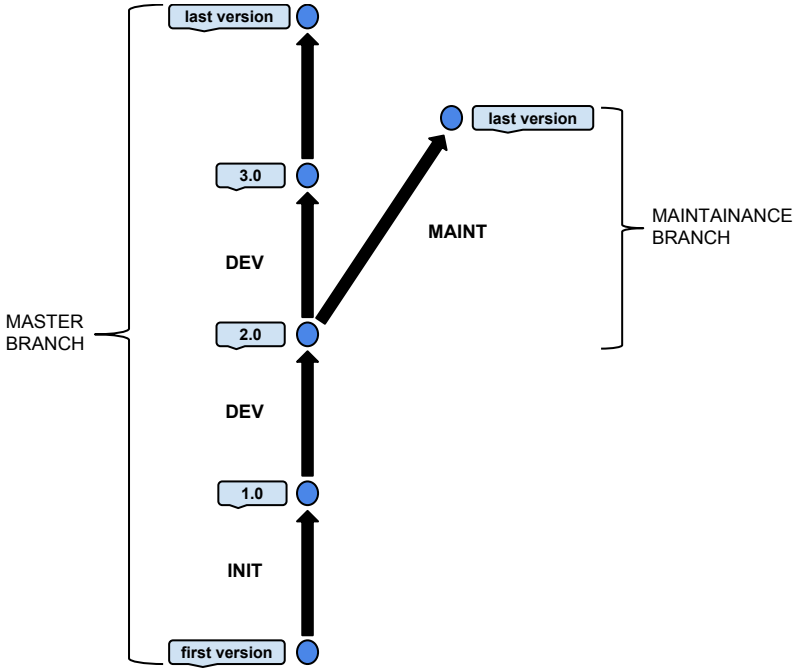

FIGURE 1 – Modèle d'architecture de dépot de code source

Nous avons donc décidés d'utiliser à partir de maintenant la detection 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.

Nous définissons maintenant le modèle Figure 1, que les projets que nous allons analyser devrons respecter.

Ce modèle représente une architecture pour le dépot de code source dérivé de l'architecture Git Flow.
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 accécible aux utilisateurs, puis cette release est maintenu 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 initial (init) entre la première version du logiciel (incluse) et la première release, puis des périodes de développement (dev) entre chaque releases. Chaque projet contient des releases majeurs, qui correspondes à des point clé du projet, des périodes susceptible de contenir beaucoup de changements, et des releases mineurs. On distingue les releases majeurs des mineurs par une augmentation significative du numéro de release. Par exemble $1.9 - 2.0$ pour Jquery, $3.7 - 4.0$ pour PHPUnit ou $0.13 - 1.0$ pour Rails. De l'autre côté les branches de maintenance sont divisés en périodes (maint) entre chaque releases.

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

TABLE 2 – Our corpus of software projects.

6 Un ensemble de projet

Nous avons donc du sélectionner un ensemble de projets sur lesquels effectuer nos expérimentations qui respectent le modèle définit. Des projets open-source, conséquents et connues de la communauté MSR. Nous avons un ensemble de projets utilisé par l'equipe de Génie Logiciel au LaBRI qui respectent le modèle avec des branches de maintenances identifiés. Les 5 projets qui sont donnés Table **??** nous fournissent un corpus pour notre prochaine expérience avec differents langages de programmation, un nombre de lignes de code ainsi qu'un nombre de développeurs dans la moyenne jusqu'à évelé par rapport aux projets open source utilisés par la communautée. Les 5 projets sont gérés sur Git afin de profiter du détectage automatique des renommages (section).
De plus, il faut noter que nous avons choisis d'exclure tout les fichiers qui ne sont pas du code source du corpus étant donné que les métriques de procédés sont habituellement uniquement calculé sur ces fichiers.

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 périodes comme décrites plus tôt sur chaque projets de notre corpus. Nous comptons donc sur la detection automatique de renommage de Git et l'utilisons dans notre outils développé en Ruby pour obtenir notre chiffres (détails dans la section résultats). (TODO pq Ruby ?) Plus précisement nous suivons ces trois étapes entre chaque périodes :

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 receuillis par exemple le pourcentage de fichiers $\%F_R$ qui ont étés renommés au moins une fois durant la période.

A notre connaissance il n'existe pas d'évaluation empirique de l'algorithm utilisé par Git. Néanmoins nous procédons à une évaluation manuelle de son comportement

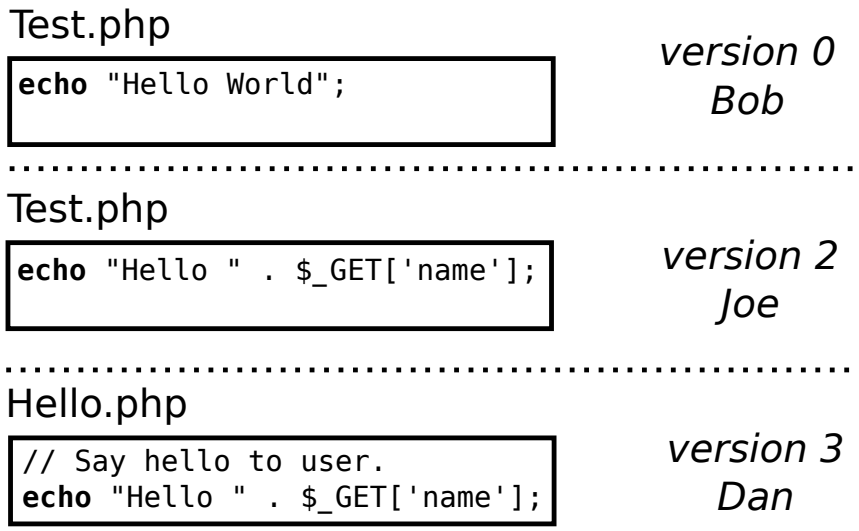

FIGURE 2 – Example of a project history. The project is composed of only one file `Test.php` which is renamed to `Hello.php` in the last version.

dans la partie conclusion et nous n'avons pas noté de faux positif sur 100 renommages aléatoire récupéré par notre outil.

7.1.1 Métriques et Renommage

Un gestionnaire de versions (VCS) offre plusieurs moyens de calculer les métriques de procédés car il stocke les informations sur les entités modifiés 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 entités modifiés lors d'une période puis de ne garder uniquement les entités toujours présente à la dernière version de notre période.
Par ailleurs, il faut noter qu'un VCS identifie une entité par son chemin + 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, on présente 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 étés identifiés comme auteurs d'une modification sur l'entité pendant la période donnée.

Si on ne prend 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'apparait que lors de la version 3. Le calcul des métriques est donc trivial, NoD = 1
Par ailleurs, si on prend en compte le fait que ce fichier a été renommé, il y a trois versions à regarder en ce qui concerne l'entité. Le premier nom du fichier était

Test.php. Ce fichier à un premier autheur lors de la version 1 puis un deuxième à la version 2. Le fichier est ensuite renommé en Hello.php par un troisième autheur. 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 précisement comment nous avons procédés :

Nous récupérons d'abord la dernière version du projet pour obtenir les entités existante à la fin de la période considéréé. On note A cet ensemble d'entités. Deuxièmement, toujours grace aux commandes git, nous récupérons toute les modifications effectués durant la période. On note C l'ensemble des modifications dans l'ordre chronologique. Troisièmement nous parcourons cet ensemble de modification en commencant 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 entités. 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'autheurs qui ont effectués 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 crée 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 ajoutés ou suprimés.

7.2 deuxième expérience

Le but de la deuxième expérience est de voir si le renommage peut biaiser significativement les valeurs des métriques de procédés décrites ci dessus. Pour ca, nous effectuons une analyse dans le pire des cas. On sélectionne la période de nos projets qui a la plus grande valeurs de fichiers renommés en excluant la période initiale qui n'est généralement pas observé dans les études. Nous calculons ensuite les trois métriques avec et sans le renommage de fichiers pris en compte, puis nous calculons la correlation de coéficient de Spearman entre les métriques avec et sans la detection de renommage. Un gros coéficient, proche de 1, indiquera que les métriques avec et sans détection de renommage sont très similaires alors qu'un coéficient plus petit, 0.5 et moins, indiquera que les métriques avec et sans détection de renommage sont très différentes.

[tabular=rccccc, table head=

Renaming metrics

Period Kind $\#F$ $\#AF$ $\%AF$ $\%F_r$ $\%AF_r$

, late after line=
, late after last line=

, after table=]data/tables/jenkins.csv1=,2=,3=,4=,5=,6=,7=

TABLE 3 – Amount and location of renaming in Jenkins

[tabular=rccccc, table head=

Renaming metrics

Period Kind $\#F$ $\#AF$ $\%AF$ $\%F_r$ $\%AF_r$

, late after line=
, late after last line=

]data/tables/jquery.csv1=,2=,3=,4=,5=,6=,7=

TABLE 4 – Amount and location of renaming in JQuery

8 Resultats

8.1 resultats première expérience

Les resultats de la première expérience sont montrés dans les tables Table **??**, Table **??**, Table **??**, Table **??** et Table **??**.

- *Number of files (#F) :* number of files in the project at the last version of the period.
- *Number of active files (#AF) :* number of files created, deleted, copied, modified or renamed during the period and present at the last version of the period.
- *Percentage of active files (%AF) :* $\%AF = \frac{\#AF}{\#F}$.
- *Number of renamed active files ($\#AF_r$) :* number of active files that have been renamed.
- *Percentage of files renamed ($\%F_R$) :* $\%F_R = \frac{\#AF_R}{\#F}$.
- *Percentage of active files renamed ($\%AF_R$) :* $\%AF_R = \frac{\#AF_R}{\#AF}$.

These tables show several interesting particularities. Firstly, the amount of renaming vary a lot between projects and release. For instance Jenkins has at most 10% of its files renamed in the worst period while PHPUnit has 98.51%. In general, there is a large amount of periods with 0% of renamed files.

Regarding the location, the init period seems to be the worst one. It generally has the greater percentage of renamed files (except in PHPUnit). The dev periods are more prone to renaming than the maint periods, as all five projects have close to 0% of renamed files in maint periods. Finally dev periods can contain a lot of renaming. The results seems to indicate that the major releases are the worse.

[tabular=rccccc, table head=

Renaming metrics

Period Kind $\#F$ $\#AF$ $\%AF$ $\%F_r$ $\%AF_r$

, late after line=
, late after last line=

]data/tables/phpunit.csv1=,2=,3=,4=,5=,6=,7=

TABLE 5 – Amount and location of renaming in PHPUnit

[tabular=rccccc, table head=

Renaming metrics

Period Kind $\#F$ $\#AF$ $\%AF$ $\%F_r$ $\%AF_r$

, late after line=
, late after last line=

]data/tables/pyramid.csv1=,2=,3=,4=,5=,6=,7=

TABLE 6 – Amount and location of renaming in Pyramid

8.2 resultats deuxième expérience

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.

[tabular=rccccc, table head=

Renaming metrics

Period Kind $\#F$ $\#AF$ $\%AF$ $\%F_r$ $\%AF_r$

, late after line=
, late after last line=

]data/tables/rails.csv1=,2=,3=,4=,5=,6=,7=

TABLE 7 – Amount and location of renaming in Rails

[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.