Impact of Artifact Renaming on Software Change Metrics

Jean-Rémy Falleri, Pierre Chanson, Matthieu Foucault and Xavier Blanc
University of Bordeaux
LaBRI, UMR 5800
F-33400, Talence, France
Email: {falleri,mfoucaul,xblanc}@labri.fr, pierre.chanson@etu.u-bordeaux.fr

Abstract

Software change metrics (also called process metrics), such as code churn or number of developers, were shown good indicators for defect prediction in several studies. However, as these metrics are computed from the changes performed on a software artifact, their values may be skewed in case of artifact renaming, which is known to confuse version control systems and can thus be a important threat to validity of studies using such metrics. In this article, we assess the impact on artifact renaming on change metrics. To that extent, we perform an empirical study on five open-source popular and mature software projects with the intent give some insight about the amount of renaming, and its effect on change metrics values. We observed that artifact renaming can significantly skew the values of change metrics in some projects. Finally we evaluate past studies with regard to this threat and provide guidelines for future studies.

1. Introduction

Defect prediction, which mainly aims to anticipate the number of defects that will be contained in a software project, and to guess their location in the source code, is one of the famous challenges of software engineering with the objective to identify the best metrics that can serve as a defect predictor [1].

In [2], Nagappan et al. show that neither classical metrics such as LOC (line of code) nor object-oriented ones such as inheritance tree (DIT) can be used as a predictor for defects in all projects. Since then, many recent studies show that software change metrics (also called process metrics) give much better results [3], [4], [5], [6].

Change metrics focus on evolution, and measure how artifacts of a software project are modified during its life-cycle. The main hypothesis behind these metrics is that the manner in which artifacts are changed has a huge impact on their quality, and therefore on defects they will contain.

In [7], Radjenovic et al. identify that the three most used change metrics for defect prediction are: the Number of Developers (NoD) [4], the Number of Changes (NoC) [8] and Code Churn (CC) [9]. NoD measures how many developers did contribute to an artifact. NoC measures how many changes have been made to an artifact. Code churn measures how many lines of code were added or removed to an artifact.

Computing change metrics seems to be straightforward at first sight. For a given software project, it consists in observing all the changes performed to each software artifact it contains. To that purpose, the use of a Versioning Configuration Systems (VCSs) is of a great help as it tracks all changes made by all developers to all artifacts. However, VCSs provide few support to artifact renaming, which makes the computation of change metrics tricky and error prone.

Artifact renaming occurs when a developer modifies the name of an artifact or when she moves it into another directory. In theory, artifact renaming does have an impact on the computation of change metrics as it shortens the life of the artifacts when it is not considered. In practice, we don't know the amount of renaming and its impact on change metrics.

This article tackles the problem of the impact of artifact renaming on the computation of change metrics. We present an in-depth empirical study of five mature and popular open-source software projects with the intent to provide some insight on the amount of artifact renaming in software projects. Additionally, we assess the impact of artifact renaming on change metrics by computing the three most used change metrics on the projects of our corpus with and without considering artifact renaming.

Our results indicate that artifact renaming do occur in projects and can be intensive. We observed up to 99% of files renamed in one project. We also observed that it can significantly alter the values of change metrics, and therefore can be a serious threat to the validity of studies using such metrics. Based on our observations, we provide a brief analysis of the possible impact of artifact renaming in past studies that used change metrics for defect prediction, and we provide simple guidelines that will help researchers and practitioners to better compute change metrics.

To sum up, this article makes the following contributions:

- An empirical study of the artifact renaming phenomenon on five mature and popular open-source software projects.
- Detailed information on the amount of renaming.
- An analysis of the impact of not taking renaming into account on the values of change metrics.
- An analysis of past studies that used change metrics for defect prediction with the intent to evaluate if they are possibly impacted by the artifact renaming phenomenon.
- Several simple guidelines aiming at reducing the threat of artifact renaming when computing change metrics.

This paper is structured as follows: Section 2 explains how change metrics are computed and presents how artifact renaming has an impact on them. Section 3 presents the detailed methodology of our study, including the construction of our corpus and the experiments we performed. Section 4 presents the results of our study which show that artifact renaming does occur and does have an impact on change metrics. Section 5 presents the possible threat of artifact renaming in past studies as well as our guidelines to limit its impact. Finally Section 6 concludes this paper.

2. Change Metrics

This section first describes how change metrics are computed. Then it explains how artifact renaming can impact change metrics. Finally, it describes how Version Control Systems (VCSs) handle artifact renaming.

2.1. Computing Change Metrics

Change metrics measure how much software artifacts have been modified during a given period of time. The approaches that use them to predict defects usually consider a period included between two releases of a software project (quite often the last release and the one before). Moreover, as they aim to predict the defects that will exist after the period, they only consider the software artifacts that are still there at the end of the period, and not the ones that have been deleted during the period. Computing change metrics then requires to fetch all the changes made during the period to software artifacts that exist at its end.

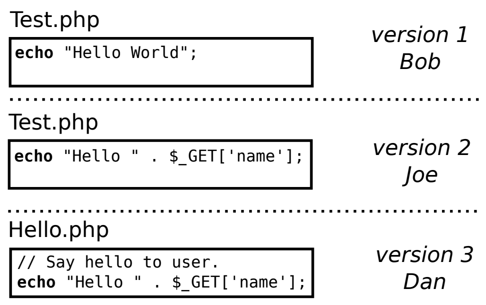
A VCS offers several facilities to compute change metrics. It stores an ordered set of versions where each version indicates which artifacts have been changed, what changes have been done to them, who did these changes, and when. Further, it supports getting the content of all the artifacts of a software project at a given version, by returning a snapshot of the software project for that version. To compute change metrics, it can therefore be used first to obtain all the artifacts that exist at the end of the considered period (by obtaining a snapshot of that period), and second to return all changes made during the period and to filter in the ones that target the considered artifacts.

More precisely, let us explain how a VCS can be used to compute the three most used change metrics: Number of Developers (NoD), Number of Changes (NoC) and Code Churn (CC). In this example, we consider as a period the one included between the last release of a software project and the one just before. To compute the change metrics, we first use the VCS to get a snapshot of the last release with the objective to get all the artifacts that do exist at the end of the considered period. We note A this set of artifacts. Secondly, we use the VCS to get all changes performed during the period. We note C this ordered set of changes. Thirdly, we iterate on this ordered set of changes starting with the older one ($c_0 \in C$) and ending with the more recent one ($c_n \in C$) in order to compute the change metrics for each of the artifact. We note μ_a^M the value of the change metrics M for the artifact a. Then we compute the change metrics as follow (we note c_i the current change we iterate on):

NoD For each artifact a targeted by c_i that also belongs to A ($a \in A$), we add to μ_a^{NoD} the number of authors that performed the change c_i and that was modifying a for the first time in the period.

NoC For each artifact a targeted by c_i that also belongs to A ($a \in A$), we add one to μ_a^C as the change c_i states that one more change has been performed.

CC For each artifact a targeted by c_i that also belongs to A ($a \in A$), we first check if the change is not an artifact creation. If it is the case it means that the artifact has been created during the period, then we initialize its μ_a^{CC} to its number of lines. Then, we use the VCS to fetch the next change c_j in the period ($i < j$) that targets a, and we then



```
echo "Hello World";
```

```
echo "Hello " . $_GET['name'];
```

```
// Say hello to user.
echo "Hello " . $_GET['name'];
```

Figure 1: Example of a project period. The project is composed of only one file `Test.php` which is renamed to `Hello.php` in the last version.

compare the two versions i and j of the artifact a to compute how many lines have been added or removed, and we add this number to μ_a^{CC}.

2.2. Renaming and Change Metrics

As we just explained, a VCS drastically helps for computing change metrics. However, it should be noted that a VCS identifies artifacts only by their qualified name (path + name). As a consequence, artifact renaming has an impact on the computation of change metrics. To explain this impact, we present a simple example of a software project period in Figure 1. This project contains only one artifact, `Test.php`, which is renamed to `Hello.php` in the last version. In this example we compute the change metrics NoD, NoC and CC, for the period included between version 1 and 3.

The last version of the period contains only one artifact. As a consequence, only this artifact will be considered by approaches that aim to predict defects. Further, if the renaming of this artifact is not taken into account to compute the change metrics, there is only one version (version 3) that targets it. As a consequence, computing the change metrics is quite trivial. As there is only one developer then $\mu^{NoD} = 1$. As there is only one change then $\mu^{NoC} = 1$. As the artifact has been created during the period then $\mu^{CC} = 2$.

However, if the renaming of this artifact is taken into account then there are 3 versions that target the artifact. As a consequence, the change metrics have completely different values. As three developers contributed to the artifact, then $\mu^{NoD} = 3$. As three changes were made to it, then $\mu^{NoC} = 3$. Finally, the code churn for the three versions is $\mu^{CC} = 4$ (1 for the creation during the period, 2 between versions 1 and 2, 1 between versions 2 and 3).

Our example shows that artifact renaming can have an impact on change metrics. More generally, the values of the NoD and NoC metrics can only be greater when the renaming is taken into account as the longer the life of an artifact, the more developers work on it, and the more changes are made to it. The value of code churn can increase or decrease if renaming is taken into account. In particular, artifact renaming induces that the renamed artifact has been created during the period. It results in adding to the CC metric the number of lines of the artifact at the version where it was renamed. Therefore, if the artifact is big, its CC will be much higher than it should be. Another interesting effect we can remark is that the closer to the end of the period an artifact is renamed, the worse the effect will be. In particular, if it is renamed just before the last version then almost all the changes that target it are lost.

2.3. Renaming and VCSs

We have performed an in-depth analysis of the main VCSs (CVS, Subversion, Git and Mercurial) with the intent to describe the mechanisms they provide to handle renaming. All these VCSs are file-based meaning that software artifacts are files, which are identified by their full path. For all these VCSs, a change in the path of a file, which can be achieved by moving it between directories or by a change of its name, leads to a file deletion and a file creation.

Some VCSs provide a mechanism to handle renaming. The renaming handling can be either manual or automatic. A renaming handling mechanism is said to be manual when the developers have to manually perform a command to indicate that their change is a file renaming. A renaming handling mechanism is automatic when the VCS provides an algorithm that can automatically detect renaming. Further, an automatic mechanism can be by default when the VCS applies the algorithm automatically when the changes targeting a file are searched for, or optional when the VCS apply the algorithm only if the user has explicitly provided some command line options.

Table 1 shows how the main VCSs handle renaming. CVS does not handle renaming. Subversion and Mercurial just provide a manual renaming handling mechanism. Git provides an automatic but optional renaming handling mechanism. Finally, we can see that no VCS provides automatic renaming handling by default. The renaming handling of these VCSs however fail to limit the impact of renaming on the computation of change metrics.

On the one hand, the manual renaming handling assumes that developers do use the command provided by their VCS. However, several studies point out that it is not the case at all. In particular, two studies observed that renaming without using the VCS command can be up to 89% of all renaming [10], [11]. Additionally, the study of Kim et al. shows that 51% of the developers prefer to perform refactoring (including renaming) without using the commands provided by their VCS [12]. These three studies, conducted in both open-source and industrial settings, show that it is very dangerous to rely on the assumption that the developers do use the VCS command provided to handle renaming.

On the other hand, optional automatic renaming handling require to have a significant knowledge of the internal settings of the VCSs, in order to use the right options to handle renaming. However, we never encountered any information explaining how the VCSs were tuned in any study that aims to predict defect by using change metrics. This is not a surprise as our analysis of past studies described in Section 5 indicates that no past study ever used a software project managed by Git, which is the only VCS that provides optional automatic renaming handling.

3. Methodology

In this section we present the methodology of our experiments that consist in studying the phenomenon of artifact renaming and its impact on the computation of change metrics. Our main objective is to evaluate if artifact renaming has a significant impact on the values of change metrics, and thus can be a threat to the validity of studies that aim to predict defects by using such metrics. To fulfill this goal, we conduct two successive experiments.

The goal of the first experiment is to analyze the amount of renaming in all the development periods of our corpus of projects. Leveraging on the results of this first experiment, our second experiment provides a worst case analysis of the impact of renaming on change metrics. For these two experiments, we use a corpus of five popular and mature open-source projects as shown in Table 2. Our corpus contains projects with different programming language and having a medium to high number of lines of code and developers. Moreover, the five projects we selected are all managed by Git because it provides a mechanism to automatically detect renaming (see Section 2). Finally, it should be noted that we choose to exclude non source code files from our corpus as defect prediction is usually only performed on source code files, therefore change metrics are computed only on such files.

	Renaming handling		
		Automatic	
VCS	Manual	Default	Optional
CVS			
Subversion	×		
Mercurial	×		
Git			×

Table 1: Handling of renaming of the main VCSs.

3.1. First Experiment

The objective of our first experiment is to better understand the phenomenon of artifact renaming. In particular, we aim to observe when renaming occurs and in which amount. To that extent, we decided to split the projects of our corpus in several periods, and to analyze artifact renaming in each period. As described in Section 2, a period is delimited by two releases and is composed of an ordered set of versions. Additionally, we distinguish several kinds of periods. Firstly, each project have a period that starts at its creation and ends at the first release. We call this period the initial period. Then the other periods can be divided into two groups: periods for a major release (major periods) and periods for a minor release (minor periods). On the one hand,major periods usually contain a large amount of changes. They happens when the release number differs significantly from the previous one. For instance the period 1.9 - 2.0 for JQuery is such a period. On the other hand, minor periods usually contain a low to medium amount of changes. They happens when the release number differs only slightly from the previous one. For instance the period 1.6 - 1.7 for JQuery is such a period. Since versioning scheme are specific to projects, we manually analyzed the repositories of the projects from our corpus to identify their minor and major periods.

To identify artifact renaming, we decided to rely on the automatic mechanism provided by Git. More precisely, we follow this three steps process to compute the amount of artifact renaming within a period:

1) list the files at then end of the period.
2) for each such file, extract its set of changes during the period by activating renaming detection (using the `git log -M` command)
3) compute the ratio of files $\%F_R$ that include a renaming in their corresponding set of changes

Git provides a dedicated algorithm to track artifact renaming based on file similarity. To the best of our knowledge, there exists no empirical evaluation on

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: Our corpus of software projects.

Git's algorithm. Therefore, we proceeded to a manual assessment of its behavior on a random subset of renamed files detected in our corpus, as described in Section 4.3. We did not encounter any false positive during this manual assessment.

3.2. Second Experiment

The goal of our second experiment is to assess if renaming can skew significantly the values of change metrics. To fulfill this goal, we perform a worst-case analysis. To that extent, we select for each project from our corpus the period that has the greater amount of renaming, excluding the initial period which is usually not used in defect prediction studies. For such periods, we compute the three change metrics described in Section 2 (NoD, NoC and CC) with and without using renaming detection. Finally, we compute the Spearman correlation coefficients between the metrics with and without renaming detection. High correlation coefficients (close to 1) would indicate that the metrics with and without renaming detection are very similar whereas low correlation coefficients (close to 0.5 and less) would indicate that the metrics with and without renaming detection are very different.

4. Results

4.1. First Experiment

The results of the first experiment are shown in Figure 2. The first important thing to notice is that most of the periods contain only a low amount of renaming. Also, the amount of renaming vary between projects. For instance Jenkins contain only a tiny amount of renaming while PHPUnit have two periods out of four with more than 50% of files renamed. Additionally, the amount of renaming varies drastically among the periods of a same project. For instance in PHPUnit the period 3.6 - 3.7 has less than 5% of files renamed while the period 3.7 - 4.0 has almost 99% of files renamed. We can also notice that initial periods (the leftmost bars in Figure 2) contain usually the bigger amount of renaming (except for PHPUnit). A final interesting finding in this experiment is that major periods do not contain necessarily more renaming than minor periods. For instance in JQuery the period 1.1 - 1.2 has more renaming than 1.9 - 2.0. However, major periods seems to often contain more renaming than minor ones: it is the case in Rails and PHPUnit, while Jenkins and Pyramid contain no major periods.

4.2. Second Experiment

The results of the second experiment are shown in Table 3. It shows that the Spearman correlation coefficients between the metrics with and without renaming highly depend on the considered period and metric. Change metrics of the Jenkins, Pyramid and Rails projects are not affected by renaming as correlation coefficients are close to 1 in every case. On the other hand, change metrics for the PHPUnit and JQuery projects can be severely impacted by renaming. On JQuery, the CC metric is resilient to renaming, but NoD and NoC are significantly impacted. On PHPUnit, all the metrics are significantly impacted by renaming. On these two projects, the metric more sensible to renaming is number of developers. In this case, it could seriously invalidate the result of a study using these metrics. Finally, we can notice that only the periods having a very high percentage of files renamed ($\%F_R$) are impacted by renaming. We can also notice that skewed change metrics were obtained in both minor and major periods.

We manually investigated the two periods that have significantly skewed change metrics values (JQuery 1.1 - 1.2 and PHPUnit 3.7 - 4.0) to understand why it happened. We remarked that in both periods, the projects directory structures were significantly modified, with several top-level directories that changed names. Therefore a very high number of files were finally transitively renamed. This is not an unusual practice, so this phenomenon might appear in any period of any project. It is interesting to remark that in both periods, this change of structure was performed in only one commit very close to the end of the period.

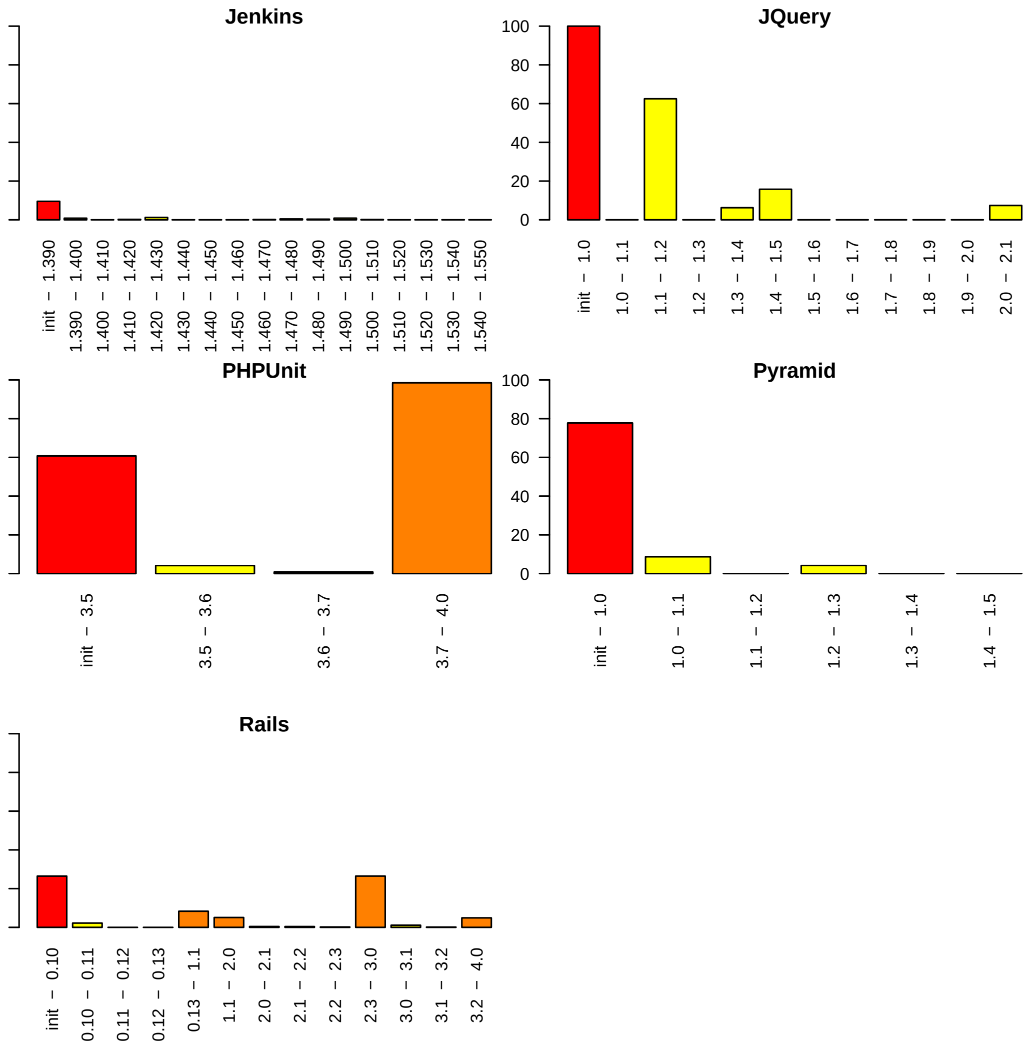


Figure 2: Percentage of files renamed ($\%F_R$) in each period of each project of our corpus. The initial period is in dark grey, major periods are in medium grey and minor periods are in light grey.

4.3. Threats to Validity

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

		Change metrics		
Period	$\%F_R$	CC	NoD	NoC
Jenkins 1.420 - 1.430	0.86	1***	1***	1***
JQuery 1.1 - 1.2	62.5	0.98***	**0.08**!	**0.59**!
PHPUnit 3.7 - 4.7	98.51	**0.6*****	**0.38*****	**0.71*****
Pyramid 1.0 - 1.1	8.69	0.97***	1***	1***
Rails 2.3.0 - 3.0.0	26.49	0.98***	0.96***	0.93***

Table 3: Spearman correlation coefficients between values of change metrics with and without renaming. The significance codes are: *** ≤ 0.01, ** ≤ 0.05, * ≤ 0.1 and ! > 0.1. Medium and low correlation coefficients are displayed in bold.

mitigate this risk we used the most used change metrics as shown in [7]. Other change metrics are often based upon these metrics (such as *code ownership* [5] or *module activity focus* [13]).

For the NoD metric, we did not apply an identity merging algorithm [14]. 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 to small to generalize our conclusion.

5. Analysis of Past Studies

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 changes 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 4, 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 [7] that use the CC, NoD or NoC metrics. However several other studies referenced in [7] use slightly different change metrics and could also be impacted on renaming.

15 past studies use industrial software projects [15], [8], [16], [17], [9], [3], [18], [19], [20], [21], [22], [23], [4], [24], [25]. All these studies did not list artifact renaming anywhere in the data collection or threats to validity sections. Unfortunately, the lack of information about VCSs and software projects used in these studies forbid us to evaluate if artifact renaming is a possible bias for their result. However, the article of Kim et al. [12] points out the fact that industrial developers can also perform refactoring (including renaming) without using the dedicated tool. Therefore these studies might be impacted depending on the tools and habits of developers involved in the industrial projects.

11 studies use open-source software projects[26], [27], [28], [29], [29], [30], [31], [32], [33], [34], [35], [36]. The VCSs used by the projects included in these studies are either CVS or Subversion. CVS do not handle renaming, and Subversion handles renaming manually which is dangerous as explained in [10], [11]. However, only 2 of these studies [34], [35] listed artifact renaming in their data collection or threat to validity sections. To mitigate the risk of artifact renaming, these 2 studies deleted from their corpus files that have been added or removed during the analyzed periods. This is an effective way of avoiding computing skewed change metric values. However, it can remove unnecessarily a significant amount of files from the corpus, which in turn might bias the study. In conclusion, all these studies might also be impacted by artifact renaming.

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: [37], [10], [11]. 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 [38]. 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.

References

[1] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, p. 675689, Sep. 1999. [Online]. Available: http://dx.doi.org/10.1109/32.815326

[2] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, p. 452461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, p. 284292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[4] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, Oct. 2008. [Online]. Available: http://link.springer.com/article/10.1007/s10664-008-9082-8

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, p. 414.

[6] E. Giger, M. Pinzger, and H. Gall, "Can we predict types of code changes? an empirical analysis," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, Jun. 2012, pp. 217 –226.

[7] D. Radjenovi, M. Heriko, R. Torkar, and A. ivkovi, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000426

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, p. 653661, Jul. 2000. [Online]. Available: http://dx.doi.org/10.1109/32.859533

[9] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Software Maintenance, 1998. Proceedings. International Conference on*, 1998, p. 2431. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=738486

[10] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou, “Inferring repository file structure modifications using nearest-neighbor clone detection,” in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct. 2012, pp. 325–334.

[11] D. Steidl, B. Hummel, and E. Juergens, “Incremental origin analysis of source code files,” *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[12] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. New York, NY, USA: ACM, 2012, p. 50:150:11. [Online]. Available: http://doi.acm.org.gate6.inist.fr/10.1145/2393596.2393655

[13] D. Posnett, R. D’Souza, P. Devanbu, and V. Filkov, “Dual ecological measures of focus in software development,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, p. 452461.

[14] M. Goeminne and T. Mens, “A comparison of identity merge algorithms for software repositories,” *Science of Computer Programming*, vol. 78, no. 8, pp. 971–986, Aug. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642311002048

[15] E. Arisholm, L. C. Briand, and E. B. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” *Journal of Systems and Software*, vol. 83, no. 1, pp. 2 – 17, 2010, SI: Top Scholars. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209001605

[16] T. Khoshgoftaar, R. Shan, and E. 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*, 2000, pp. 301–310.

[17] L. Layman, G. Kudrjavets, and N. 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*, ser. ESEM ’08. New York, NY, USA: ACM, 2008, p. 206212. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414038

[18] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality: an empirical case study,” in *Proceedings of the 30th international conference on Software engineering*, 2008, p. 521530.

[19] N. Nagappan and T. 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*, ser. ESEM ’07. Washington, DC, USA: IEEE Computer Society, 2007, p. 364373. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2007.87

[20] N. Nagappan, T. Ball, and B. 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*, ser. ISSRE ’06. Washington, DC, USA: IEEE Computer Society, 2006, p. 6274. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2006.50

[21] 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*, Nov. 2010, pp. 309–318.

[22] A. P. Nikora and J. C. Munson, “Building high-quality software fault predictors,” *Software: Practice and Experience*, vol. 36, no. 9, p. 949969, 2006. [Online]. Available: http://dx.doi.org/10.1002/spe.737

[23] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Programmer-based fault prediction,” in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE ’10. New York, NY, USA: ACM, 2010, p. 19:119:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868357

[24] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, “Using developer information as a factor for fault prediction,” in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, p. 8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1269056

[25] X. Yuan, T. Khoshgoftaar, E. 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*, 2000, pp. 85–90.

[26] 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*, Oct. 2009, pp. 135–144.

[27] A. Bacchelli, M. D’Ambros, and M. Lanza, “Are popular classes more defect prone?” in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE’10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 5973. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12029-9_5

[28] B. Caglayan, A. Bener, and S. 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*, ser. FLOSS '09. Washington, DC, USA: IEEE Computer Society, 2009, p. 3136. [Online]. Available: http://dx.doi.org/10.1109/FLOSS.2009.5071357

[29] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, p. 531577, 2012. [Online]. Available: http://link.springer.com/article/10.1007/s10664-011-9173-9

[30] 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*, May 2010, pp. 31–41.

[31] T. Illes-Seifert and B. Paech, "Exploring the relationship of a files history and its fault-proneness: An empirical method and its application to open source programs," *Information and Software Technology*, vol. 52, no. 5, pp. 539–558, May 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909002109

[32] P. L. Li, M. Shaw, and J. 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*, 2005, p. 32.

[33] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, 2010, p. 18.

[34] R. Moser, W. Pedrycz, and G. 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*, ser. ESEM '08. New York, NY, USA: ACM, 2008, p. 309311. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414063

[35] ——, "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*, 2008, p. 181190. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4814129

[36] A. Schrter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk," in *Proceedings of the 5th international symposium on empirical software engineering*, vol. 2, 2006, p. 1820. [Online]. Available: http://www.st.cs.uni-saarland.de/publications/files/schroeter-isese-2006b.pdf

[37] 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*, Sep. 2004, pp. 31–40.

[38] W. Wu, Y.-G. Guhneuc, G. Antoniol, and M. Kim, "AURA: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, p. 325334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848