Impact of Renaming on Software Change Metrics

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

***Abstract*—Software change metrics, such as *code churn* or *number of developers*, are good indicators for software quality, as it has been observed in several studies. However, as these metrics are computed from the changes performed to a software artifact, their value may be skewed in case of artifact renaming, which can therefore be a important threat to validity. This issue is underestimated in most of the existing software study. We therefore propose to assess its impact in this paper. To that extent, we have performed an empirical study on five open-source programs with the intent to evaluate the amount of renaming and their effect on change metrics. We observed that renaming can significantly alter the metrics for some projects. We also noticed that this effect can be minimizing by following some simple guidelines presented in this paper.**

I. Introduction

Software *change metrics* measure how the artifacts of a software project have been modified during their life-cycle. For instance, the *code churn* measures how much lines have been added or removed, and *number of developers* measures how many developers did contribute to the artifact. The main hypothesis behind these metrics is that the manner in which the artifacts have been developed has a huge impact on their quality. Such an hypothesis have been confirmed in several studies. For instance, Bird et al. have shown that the ownership metrics, which are variants of the number of developers metric, are strongly correlated to the number of post-release defects in industrial software projects [**?**].

Contrarily to classical product metrics (such as *number of methods*) which only require one version of a software project, change metrics require its history. Generally, software artifacts are seen as a sequence of versions, and change metrics are computed from measures performed on each version of the sequence. Generally, computing change metrics is straightforward when the projects are managed by a version control system (VCS), which is almost always the case.

However, retrieving the complete sequence of version of a software artifact is not always easy. Indeed, it is not rare that developers change the artifacts name during the project life-cycle. When such a situation occurs, relating the versions renamed artifact with the versions before its renaming is not easy. If the renaming is not taken into account, all its versions before renaming are lost and it can therefore have a huge effect on the change metrics values. For example, if we imagine a project where all the artifacts have been renamed in the middle of its life-cycle, then needless to say that not considering this information will return skewed measure for the code churn or number of developers metrics.

In theory, artifact renaming does have an impact on change metrics. In practice, how many renaming occurs? When do they occur? What is their impact on change metrics? This paper aims at answering these questions. To that extent, we perform an in-depth study of five real open source software projects. Our objective is first to provide descriptive statistics helping to understand the amount and location of artifact renaming in software projects. Then we assess the impact of taking into account renaming on the values of classical change metrics. Finally, based on our observations, we provide guidelines that will help researchers and practitioners to better compute change metrics.

Our results indicate that artifact renaming do occur in projects and can be intensive. We observed up to 99% of renamed files in one project. We also observed renaming are most likely to happen in software development rather than in software maintenance. Finally, we also observed that it can significantly alter the values of change metrics, and therefore is a serious threat to the validity of studies using such metrics.

To sum-up, this article makes the following contributions:

- A study of the artifact renaming phenomena made on five mature and popular open-source software projects.
- Detailed information on the amount and location of renaming.
- An analysis of the impact of taking into account renaming on the values of change metrics.
- A set of guidelines aiming at reducing the threat of artifact renaming when computing change metrics.

This paper is structured as follows: Section **??** explains how change metrics are computed and presents how the artifact renaming has an impact on them. Section **??** presents the detailed methodology of our study, including the construction of our corpus and the different analyses we realized. Section **??** presents the main results of our study which shows that the artifact renaming phenomena does occur and does have an impact on process metrics. This section also presents our guideline to limit the threat of process metrics. Finally, Section **??** of our study and Section **??** concludes this paper.

II. Change Metrics

In this section, we give some background information on change metrics, and how there are computed using version

Test.php

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

Version 0
Bob

Test.php

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

Version 2
Joe

Hello.php

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

Version 3
Dan

Fig. 1. 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.

control systems. Then, we explain using a concrete example how renaming can affect the values of these metrics.

A. Computing Change Metrics

As explained in Section **??**, change metrics are computed from the sequence of versions of a software artifact. Since projects are almost stored in version control systems (VCSs), we described how the software artifacts are stored in such VCSs. VCSs generally consider a project as a set of files. It stores every version of the project since the beginning of its development. Several metadata are attached to each version: the identity of the developer responsible for its development, the files that have been added, modified or deleted since the previous versions, and its date.

To recongnize the files across the versions, VCSs use their absolute path in the repository. It means that as soon as a file is moved or change name, the VCS will detect it as the deletion of the file with the old absolute path, and the addition of the file with the new absolute path. Altough several VCSs provide commands or algorithms to record renamed and moved files, none of them handle this problem out of the box.

B. Renaming and Change Metrics

Figure **??** shows an example of a simple software project history. This project contains only one file, `Test.php`, which is renamed in the last version to `Hello.php`. As an example of change metrics we will use *code churn*, *number of developers* and *number of modifications*. The definition of number of developers and number of modifications is straightforward. Code churn correspond to the number of lines added and deleted in the file since its initial version.

If renaming is not taken into account, the last version of the project of Figure **??** contains only one file, that has only one version. Therefore its code churn is 2 (it has 2 lines of codes), and the number of developers and modification is 1. However, if renaming is taken into account, the last version of the project contains one file that has 3 versions. In this case the code churn is 4 (1 for version 1, 2 for version 2 and 1 for version 3). The number of developers and modifications are 3.

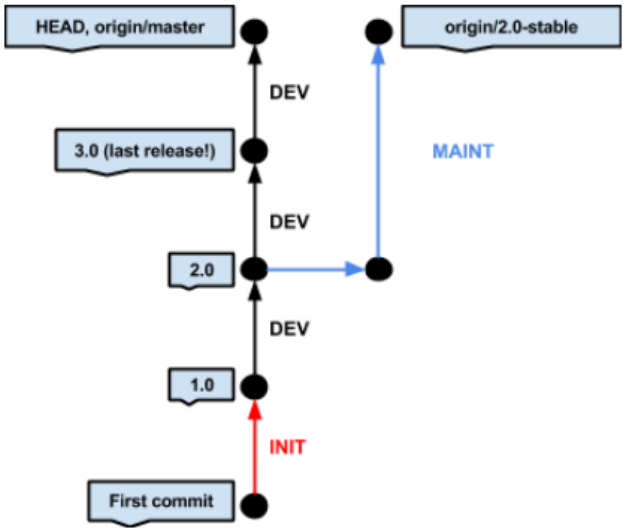

Fig. 2. Our location kinds. *Maint:* commmits from a maintenance branch. *Init:* commits from the first version (included) to the first release tag on the master branch. *Dev:* commits between two succesive release tags on the master branch.

Therefore this small example shows the importance of taking renaming into account.

III. Methodology

In this section we present the methodology of our experiment. We want to answer the following research questions:

1) Where in the history of projects is located the renaming, and what is its typical amount?
2) Is the renaming able to skew significantly the values of change metric?

To answer these questions we conduct two successive experiments. In these experiments, we will use a dataset of five popular and mature open-source projects.

A. Location and Amount of Renaming

Software projects usually undergo distinct phases in their life-cycle. Usually the development is started, then a version is released to the public and maintained, while a new version is being prepared, and so on. Therefore we want to assess the amount of renaming in these different phases. To that extent we split a project history in various *locations*. A location is a sequence of commits. Additionnaly, each location has a *location kind* as described in Figure **??**.

For each location we compute the following metrics:

- *Number of files ($\#F$):* number of files in the project at the last version of the location
- *Number of active files ($\#AF$):* number of files created, deleted, copied, modified or renamed that are still present at the last version of the location.
- *Number of renamed active files ($\#AF_r$):* number of active files that have been renamed.

- *Percentage of active files ($\%AF$):*

$$\%AF = \frac{\#AF}{\#F}$$

- *Number of modification operations ($\#MO$):* number of modifications (create, delete, copy, rename, modif) on files recorded in the location.
- *Number of rename operations ($\#RO$):* number of renames performed on the location.
- *Percentage of rename operations ($\%RO$):*

$$\%RO = \frac{\#RO}{\#MO}$$

- *Percentage of files renamed ($\%F_R$):*

$$\%F_R = \frac{\#AF_R}{\#F}$$

- *Percentage of active files renamed ($\%AF_R$):*

$$\%AF_R = \frac{\#AF_R}{\#AF}$$

B. Process

We will describe here the process to follow to obtain this numbers considering any git repository which follows our model. A Ruby script implementation of the following process is given in annexe.
First of all we need to list major releases tags in a chronological ascending order. This part can not be automated because of the different tag conventions between projetcs like:

- 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.

Begin the process:
We list the project remotes branches. Browsing the branches, all of them we be considered as maintenance branch except origin/master, the initial and developement branch. Most of the time, specific maintenance branches are followed by stable. But if the branch is listed here, it means it has its head out of master branch and so it is a maintenance branch since its last merge with master.

The work is divided in two major step:

- The maintenance branches
- origin/master

If the branch is origin/master, the work will be divided again in:

- first commit(included) to first release tag
- fisrt tag to last tag (one release after another)
- last tag to HEAD

So in any of the four conditions, we will generate the git log between revision range. The major part of the work will be on this commits log. We choosed to work on a general log more than mining the history of every files one after the other for performance reasons. Especially on big projects (rails, jenkins). Moreover this technique only works for the files alive at the HEAD of the project and not from one previous revision. So we analyse all the log in one block, with the good command and options it contains enough informations. Log lines stored in simple structures like arrays, we can easily count modifications or renames detected.
We will also need to get all the alive files at the end of the log, at the second revision range (branch head or the end of one release).

Then the principal algorithm will browse the log :
Browsing the log in a chronological order will allow us to follow the renames or modifications and rebuilt the history of files. The typical rename in the log looks like "rename bob/{henry ⇒ josef}/george.py (86%)" As exemple, if "bob/henry/george.py" is already recorded, we need to follow "bob/josef/george.py" instead in the renames and modifications record. There is some other cases to consider, for exemple "{henry ⇒ }" or without "{" or the "copy" etc. Each files recorded are unique and only the last name of renamed files is considered. We finally need exclude all the files deleted before the end of the commits log comparing with the list of alive files at the end of the revision range calculated earlier.

IV. Study

V. Results

Version	Type	Number
1.0	DEV	12
2.0	DEV	13

VI. Threats to validity

VII. Related Work

VIII. Conclusion