Analysis of Code Clone Ratios over Version Evolution in Open-Source Projects Written in C and C++

Anfernee Goon[1,†1,a)] Yuhao Wu[2,b)] Makoto Matsushita[2,c)] Katsuro Inoue[2,d)]

Abstract: A code clone is a fragment of code which is duplicated throughout the source code of a project and have been shown to make a project less maintainable because all code clones will share potential bugs and problems. This study analyzes the code clone ratios over the entire development lifetime of Git, a widely used open-source project written in C/C++ to understand development habits and the changing maintainability of the software. The study utilizes bash scripting in conjunction with CCFinderX and GitHub to automate the detection of clones across development history. The results showed very stable ratios across development history, with the code clone ratios only fluctuating greatly during the beginning of development mostly, which can imply design choices not being concrete during the beginning of development as well as considerably more functionality being added at the beginning of development relative to the rest of the development cycle. Overall, the clone ratios over the development of Git has given some insight on the different aspects of the development process such as refactoring and how Git handles such aspects. Developers should be able to improve on their approach to development and increase their software's maintainability by looking at code clone ratios over the version evolution of their own projects.

Keywords: Clone ratios, refactoring, maintainability, software development

1. Introduction

A code clone is a duplicated fragment of code. Having many code clones in a project makes it much less maintainable because all of these code clones will share potential bugs and problems [4]. These problems will propagate throughout the software with continued use of the problematic code clone fragment, and a fix for this bug may have to be applied to every one of these fragments present in the code. In this study, we analyze the code clone metrics of software over the entire development process. The changing maintainability of software over development can be tracked through such an analysis. In turn, many development habits and a greater understanding of the software development process is possible, such as the frequency of code cleanup to improve maintainability, also known as refactoring. In this paper, we seek to understand software development through the analysis of code clone metrics throughout the entire development process of Git, an open source version control system primarily written in C/C++. More specifically, we will investigate the following research questions in our analysis: **RQ1.** *How do the code clone ratios throughout development characterize development of Git?* and **RQ2.** *What does this code clone characterization indicate about software development in general?*.

[1] Department of Computer Science and Engineering, University of California San Diego
[2] Graduate School of Information Science and Technology, Osaka University
[†1] Presently with Osaka University
[a)] agoon@ucsd.edu
[b)] wuyuhao@ist.osaka-u.ac.jp
[c)] matusita@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

2. Approach

We decided to use Git for our analysis because it is a widely used software which is known to be well developed which should lessen the chance of it producing abnormal data. Along with this, it appeared on GitHub, allowing easy access to version breakpoints (commits) to analyze. Git starts at around 950 lines of code, and grows to around 200000 lines of code over around 40000 commits (we analyze about 14000 of those commits).

To detect clones and clone metrics in the source code, we used CCFinderX, a token-based clone detector [2] [4]. Using bash scripting, we automated the use of CCFinderX on every important commit in the master branch of Git by using git log with the first-parent flag. This allows for a set of commits which linearly trace Git's development. For each commit, we collected number of C/C++ files, total lines of code (LOC), total lines of code not including whitespace or comments (SLOC), total code clone lines (CLOC), as well as the tag of the commit if applicable. Along with these metrics we collected the actual clone ratios of each commit, including the clone ratios including whitespace and comments (CCR) and the clone ratios not including whitespace or comments (CVRL). These ratios are derived from the lines of code metrics, with CCR being CLOC divided by LOC, and CVRL being CLOC divided by SLOC. The scripts were designed to exclude test and example files whenever possible in order to keep analysis limited to functionality related files, and only includes .c and .cpp files. Header files are not included because most header files will be similar, and may be picked up as false positives by CCFinderX. The minimum number of tokens that a fragment needs to be considered a clone is 50 in our study.

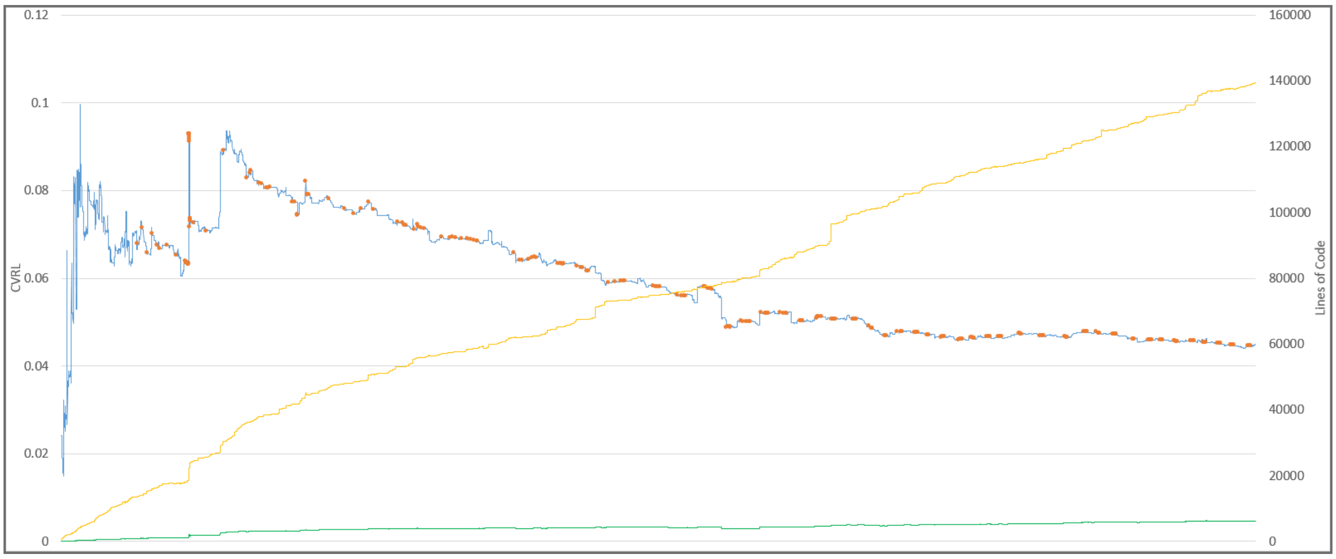

Fig. 1 CVRL, SLOC, and CLOC changes over all commits of Git in chronological order. The blue line represents CVRL, with the orange points along it displaying release points. The yellow line represents SLOC and the green line represents CLOC. The CVRL adheres to the left axis, while the other two metrics adhere to the right axis.

3. Results

For our analysis, we make use of a graph containing the metrics CVRL, SLOC, and CLOC discussed in Section 2 displayed by commits in chronological order. The main metric we focus on is CVRL, which we initially expected to have mostly gradual increases with periodic sharp declines. The gradual increases would be a result of functionality being added over time, which naturally increases CVRL because more code is being written so more clones will be introduced [1]. Refactoring would be the cause of the sharp declines, because the initial additions of functionality may not be clean and would be in need for maintenance to ensure maintainability before the next round of functionality is added.

Unlike our initial expectation, Git's CVRL sees a large growth towards the beginning of development, but after a certain point sees a gradual but consistent decrease up to the present state of development. After its large growth over around 2000 commits, the CVRL is around 9%. The gradual decrease sees the CVRL decrease to 4% over the course of about 8000 commits, and afterwards there is stability near 4% until the present state of development. The initial growth can be attributed to many additions of functionality at the beginning of development. The large amount of fluctuations at the beginning may also be a result of design choices not being concrete, causing refactoring to be needed more often and sloppier code to be written. After that initial stage, the CLOC barely increases while the SLOC continues to grow at a fast rate, which is what causes the gradual decline in CVRL, which indicates functionality is still being added. The gradual decrease may be due to better code being written or code being refactored before being committed, thus having the CVRL shrink and the CLOC grow only to maintain the 4% CVRL during the stable period. Even at its peak, Git's CVRL is significantly smaller than the average CVRL of 12% [3], which may also be due to code being refactored before commits. We also analyzed two other open-source C/C++ projects and found a similar trend of initial instability followed by a very stable period of clones ratios and overall low CVRL.

4. Conclusion

Git represents a very ideal development situation based on the CVRL graph, where the ratios are consistently very low throughout continual development. Developers can analyze their own software in a similar manner to discover how close they are to this ideal development structure. Depending on the style of development, good refactoring practices which Git's results infer may not always be possible. The code clone ratios can still be of use in determining the best frequency with which to refactor based on resource constraints. An efficient way to construct code to avoid excessive refactoring is also possible by looking at past projects mistakes in the form of these clone metrics. Assumptions aside, looking at the code clone ratios over version evolution does give insight on how well a developer is maintaining their project. This data can also help a developer improve on certain aspects of the development process by allowing them to analyze instances of large CVRL changes, understand what these changes meant to the project during past development phases, and utilize this past data to make better development decisions in the future.

References

[1] Dagenais, M., Merlo, E., Laguë, B. and Proulx, D.: Clones Occurence in Large Object Oriented Software Packages, *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '98, IBM Press, pp. 10– (online), available from ⟨http://dl.acm.org/citation.cfm?id=783160.783170⟩ (1998).

[2] Kamiya, T.: CCFinderX, http://www.ccfinder.net/.

[3] Koschke, R. and Bazrafshan, S.: Software-Clone Rates in Open-Source Programs Written in C or C++, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 3, pp. 1–7 (online), DOI: 10.1109/SANER.2016.28 (2016).

[4] Sheneamer, A. and Kalita, J.: Article: A Survey of Software Clone Detection Techniques, *International Journal of Computer Applications*, Vol. 137, No. 10, pp. 1–21 (2016).