Evolution of Code Clone Ratios throughout Development History of Open-Source C and C++ programs

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. Code clones have been shown to make a project less maintainable because all code clones will share potential bugs and problems. Unlike other code clone research, this study analyzes the code clone ratios over the entire development lifetime of three open-source projects 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 Git to automate the detection of clones across development history. The results from each project showed very stable ratios across development history, with the code clone ratios only fluctuating greatly during the beginning of development mostly and very little refactoring occurring. Despite this, the clone ratios of all three projects were very low compared to the average ratio of 12%, which indicates that refactoring may have occurred before the code was inputted into the version control repository. Each project had a general trend of fluctuating greatly at the beginning of development and then becoming very stable afterwards, 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 each project analyzed has given some insight on the different aspects of the development process such as refactoring and how each project 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 [5]. 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. Detecting code clones and retrieving different metrics about the code clones present in a target software gives insight on the maintainability of the software and identifies areas in the code that should be fixed.

Code clone research mostly focuses on the code clone metrics of a particular snapshot of a target software. Instead of focusing on a specific point in time, 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 and maintenance to improve readability and 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 three different open source projects 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 the target software?* and **RQ2.** *What do these code clone characterizations 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

2.1 Target Source Code

When deciding on open-source projects to use, we took into account a couple factors. These factors were how well studied the project is, how large the project is, and the type of the project itself. We wanted to use three projects which varied by these metrics in order to analyze the effects of these factors on the code clone ratios and what they imply about the development process. We analyzed the following three open-source projects which are hosted on GitHub:

- **libcurl:** libcurl is a library which curl, a command-line tool for transferring data, uses. This library is fairly well studied as seen in Kawamitsu et al. [3]. It is the smallest project we studied, starting at around 2500 lines of code, eventually

growing to around 12500 lines of code over a series of about 20000 commits.

- **Skynet:** Skynet is a lightweight online game framework which is slightly larger than libcurl, growing from around 2500 lines of code to around 40000 lines of code over a series of about 1000 commits (we analyze about 800 of those commits). While not as well studied as the other two projects, it is as popular as libcurl on Github based on forks and stars.
- **Git:** Git is a version control system which is widely used; even our analysis relies on Git. Although starting relatively small at 950 lines of code, it grows to around 200000 lines of code over a series of about 40000 commits (we analyze about 14000 of those commits) which is significantly larger than both libcurl and Skynet.

Each of the three open-source projects present a different range in size which makes our analysis size-independent, disregarding extremely large systems or software. Having software which is popular in use and analysis lessens the chance of abnormal data coming from one of the projects. The varying functions of each project ensure that the analysis covers the development of all types of software.

2.2 Clone Detection

To detect clones and clone metrics in the source code, we used CCFinderX, a token-based clone detector [2] [5]. Although CCFinderX is a multilinguistic clone detector, we only use it's C/C++ clone detection capabilities. Using bash scripting, we automated the use of CCFinderX on every important commit in the master branch of each project by using git log with the –first-parent flag. The commits retrieved with this flag are important because they represent the most linear development history available by following the first parent down Git's history ensuring that parallel development on separate branches are limited to the merge commit into master, which is why the entire commit range does not appear in our data. Similarly, we automate the retrieval of clone metrics from CCFinderXs results to streamline collecting the results from each commit. 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.

3. Results

For our quantitative analysis, we make use of several graphs containing the metrics CVRL, SLOC, and CLOC discussed in Section 2.2 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 [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. The results from all three projects did not quite follow this trend, and in fact all projects displayed various different patterns over development.

3.1 libcurl

Our analysis of libcurl is confined to the files in libcurl's src folder in addition to the test and example constraints previously mentioned. The src folder is the primary folder for development for libcurl's C/C++ modules, so it contains the most relevant information about libcurl's development process.

3.1.1 Quantitative Analysis

libcurl's CVRL graph details a series of sharp increases followed by gradual decreases, which is opposite of expectation. In hindsight, this finding makes sense because over the span of 20000 commits, the number of consecutive commits which add functionality will be significantly less than the total number of commits resulting in sharp increases in CVRL on the graph when functionality is added. Although decreases are still occurring, the gradual nature of these decreases indicates that refactoring is not necessarily the cause. From a perspective focused on clones, refactoring would be indicated by a decrease in CLOC and SLOC, because this means that code clone fragments are being removed. Looking at the SLOC and CLOC metrics during these periods of decrease, we observe that the SLOC is continually increasing while the CLOC does not decrease. Since CVRL is a ratio between CLOC and SLOC, it is clear that it is the growth in SLOC which is causing the decreases and not a result of refactoring.

While this trend holds true for the majority of libcurl's development, the beginning of libcurl's development does have some sharper decreases which are a result of CLOC decreasing. These decreases are not due to refactoring, which is discussed in more detail in the next section on the qualitative analysis of libcurl. Contrary to the sharp increases depicted on the CVRL graph, the CVRL is actually quite stable with the sharp increases actually only being about a 1% difference. Throughout the entire lifetime of development, the CVRL mostly falls between 3% and 9%. The average clone rate is about 12% for clones with a minimum of 100 tokens [4], so libcurl's CVRL is significantly lower than average even with a more lenient minimum token size. The stable, low CVRL of libcurl throughout development may indicate a good development habit of refactoring before commits. Refactoring before committing will ensure less clones

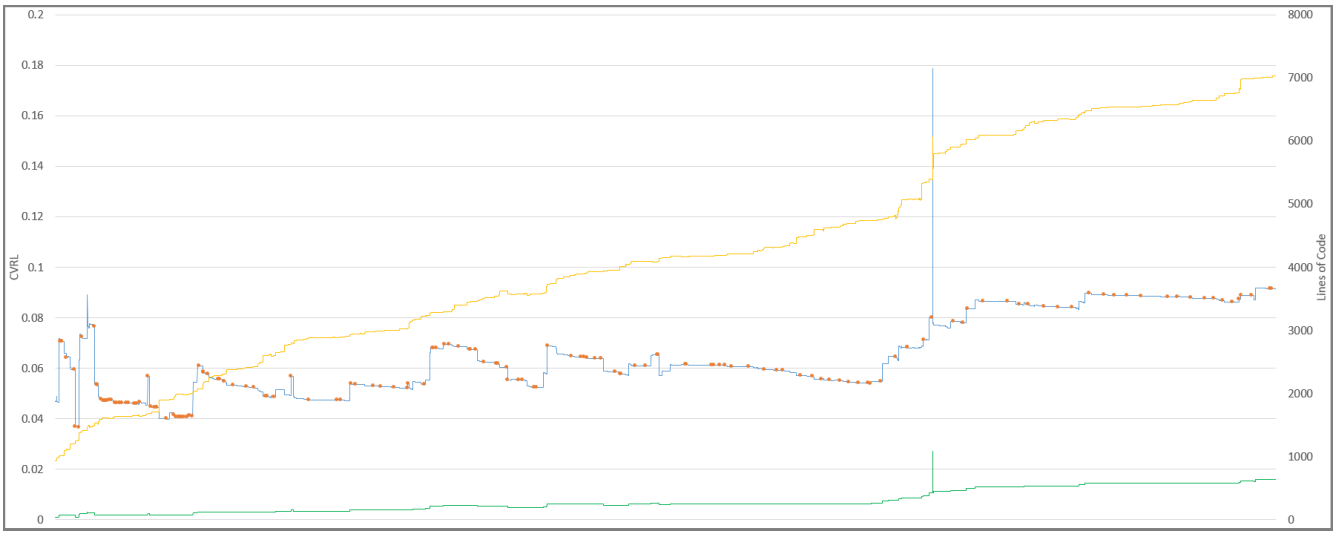

Fig. 1 CVRL, SLOC, and CLOC changes over all commits of libcurl in chronological order. The blue line represents CVRL, with the orange points along it displaying release points. The yellow line represents SLOC and the green line represents CLOC. The CVRL adheres to the left axis, while the other two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

being introduced into the git history and in turn keep the clone rate in check for each commit. The advantages of this practice are that the git history will be very clean and the software will be consistently easy to maintain. Although libcurl does have relative stability throughout its development, it is still consistently growing especially towards the end of development, where a 1000 commit sequence adds about 2% to the CVRL, but this growth still keeps the CVRL under the average.

3.1.2 Qualitative Analysis

For libcurl's qualitative analysis, we studied key rises and falls on the CVRL graph attributed to CLOC changes in further depth to see what their cause was. In order to do this, we used a tool called GemX, which is an improved version of Gemini [6] which utilizes CCFinderX with GUI options. GemX's biggest asset is it's ability to show the clone pairs found in the source code. Using this tool, we were able to analyze a big increase or decrease in CVRL and see what was the cause of the change by looking at the specific commit causing the change as well as the previous commit. There are four commit points which are looked at in libcurl: 6562caf, 22d8aa3, b5fdbe8, a0d7a26 (shortened commit hashes).

The first two commits saw a CVRL decrease of about 2% and a CLOC decrease of about 30 lines. In both these instances, the logic of the clone fragments were changed with a few additions or deletions. These fragments can still be considered code clones, but are now Type-3 code clones instead of Type-2 code clones, where Type-3 clones have some line additions and deletions which do not occur in the other fragment [5]. Since CCFinderX cannot detect Type-3 code clones, this was seen as a decrease in CLOC resulting in the CVRL decreases [5]. It is clear that the decreases were not due to refactoring in this case, but rather something more akin to a bug fix to make the fragments work correctly by changing a few parts of the fragments (and even still the fragments are Type-3 clones). This case illustrates the importance of also looking at the change in SLOC to determine if refactoring occurred, because if SLOC also decreases by a similar margin then there is a good chance the clone was removed. This phenomenon of clones becoming undetectable due to change in type is a threat to validity discussed in Section 3.5.

Unlike the first two commits, the last two commits show the most refactoring seen in libcurl. The first of the two commits, b5fdbe8, shows a huge increase in CLOC (from around 4% to around 18%) while the other commit, a0d7a26, shows a symmetrical decrease (from around 18% to around 4% again). The first commit adds a new experimental functionality through a new file which is an exact copy of another source file that is about 600 lines, increasing the CLOC by that margin. The second commit significantly changes the new source file to reuse parts of the original source file along with new features, this eliminating the entire clone fragment as it was in the previous commit and reducing the CLOC down once more. This shows the only clear case of refactoring we have found libcurl, with the developer implementing reuse of source files indirectly rather than keeping directly copied pasted code.

From our qualitative analysis, we see that even the biggest decreases in CVRL are mostly not due to refactoring, with one exception. It can be deduced from these findings that libcurl's developers most likely have a good habit of refactoring before commits, because even though there is only one case of refactoring throughout the development history the CVRL is still very low.

3.2 Skynet

Unlike libcurl's analysis, our analysis of Skynet takes account all C/C++ files that the project contains excluding files in the test folder. There was no single folder consolidating all of Skynet's C/C++ modules, so analyzing all the files in the project was a necessity.

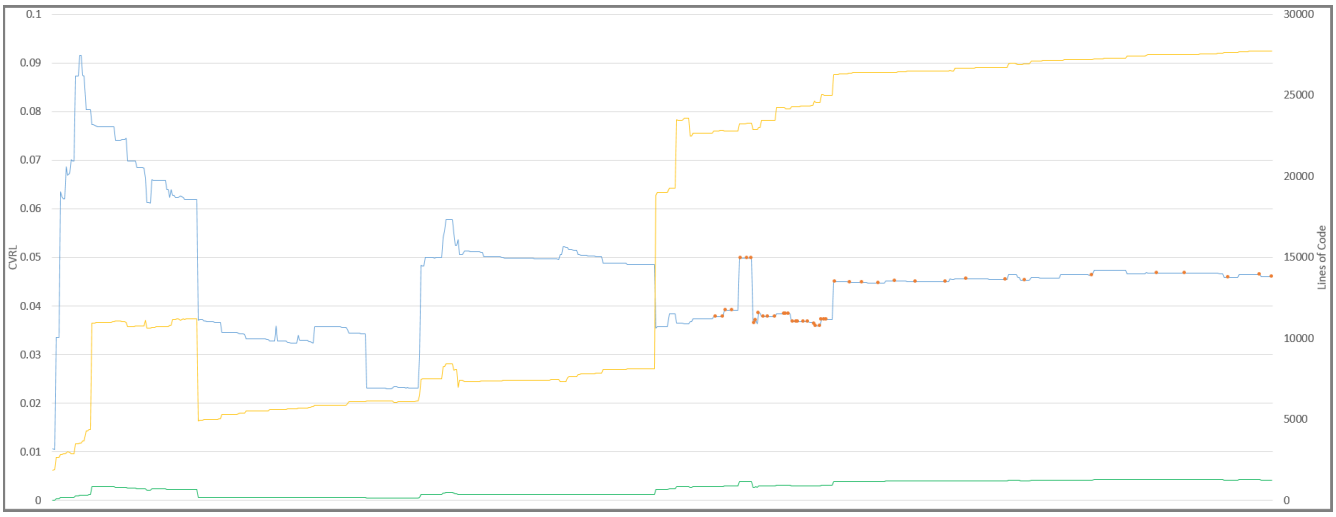

Fig. 2 CVRL, SLOC, and CLOC changes over all commits of Skynet in chronological order. The blue line represents CVRL, with the orange points along it displaying release points. The yellow line represents SLOC and the green line represents CLOC. The CVRL adheres to the left axis, while the other two metrics adhere to the right axis.

3.2.1 Quantitative Analysis

Skynet's CVRL graph partially resembles what was initially expected, but still has significant differences. The resemblance is in the graph's sharp declines, which occur only a few times throughout the entire commit history. The unexpected differences are the sharp increases which mirror the sharp declines, as well as the extreme stability which characterizes the second half of the development history. Similar to libcurl, Skynet's CVRL graph's sharp increases most likely indicate an addition in functionality. These only occur during the first half of development, but are considerable increases, with the largest increase going from 6% to around 9%. With each big increase there is a sharp decrease, which may indicate refactoring after large functionality additions. Most of these sharp decreases have corresponding CLOC decreases, but only a few have corresponding SLOC decreases as well indicating refactoring is not necessarily the case for every instance. The qualitative analysis of Skynet in the next section confirms refactoring efforts, showing that refactoring does occur in some of these instances.

Although the development of Skynet has fairly sharp fluctuations during the beginning of its development, it is akin to libcurl through its smaller than average CVRL and stability. Skynet actually shows greater stability than libcurl after its initial fluctuations, occurring directly after the first release. The stability after release seems to be logical, as a polished product should be delivered at release, reducing the additions to mostly bug fixes which are less likely to introduce new clone additions.

3.2.2 Qualitative Analysis

Skynet's qualitative analysis used the same approach as libcurl's, where we looked at big fluctuations on the CVRL graph that resulted from CLOC changes in detail using GemX. The main commits that we looked at in Skynet are 28dc840 and 58aa755.

The first two commits are decreases in CVRL of about 1% and 2% respectively. The cause of these decreases can both be attributed to changes in libraries which Skynet was using. The first commit sees a removal of a part of a library which contains 7000 lines of code, 500 of those being clone lines. Although not a refactoring of their own modules, this library removal is in fact an instance of refactoring because it removes unnecessary code, potentially opting for reuse like in libcurl's instance of refactoring which in turn reduces CLOC. The second commit adds on a large library of code to the project which proportionally has very little code. Through this addition, the CLOC only increases slightly, by about 260 lines, while the SLOC increases significantly, by about 10000 lines. Unlike the first commit, this is not a result of refactoring as none of the originally existing clone pairs were removed.

Our qualitative analysis of Skynet shows many of the decreases being attributed to library changes. Since Skynet's stable period saw much less growth, it can be inferred that most functionality was added during the initial unstable period (reinforced by the SLOC only gradually increasing during the stable period). If this is the case, it is much more unlikley that Skynet's developers employ refactoring before commits because the time where refactoring would be needed to keep the CVRL stable showed great instability and there was evidence of refactoring after commits during this period. Out of all three projects, Skynet shows the most resemblance to our initial hypothesis, but only when analyzing the beginning of its development.

3.3 Git

Similar to the analysis of Skynet, our analysis of Git takes into account all C/C++ files that the project contains excluding test and example files. Again like Skynet, Git's C/C++ modules were dispersed all throughout the project which is why all files in the project needed to be analyzed. Unfortunately all analysis of Git was done quantitatively, as a qualitative analysis was difficult given the size of the project. We hope to perform a qualitative analysis on Git in the future to reinforce our quantitative analysis.

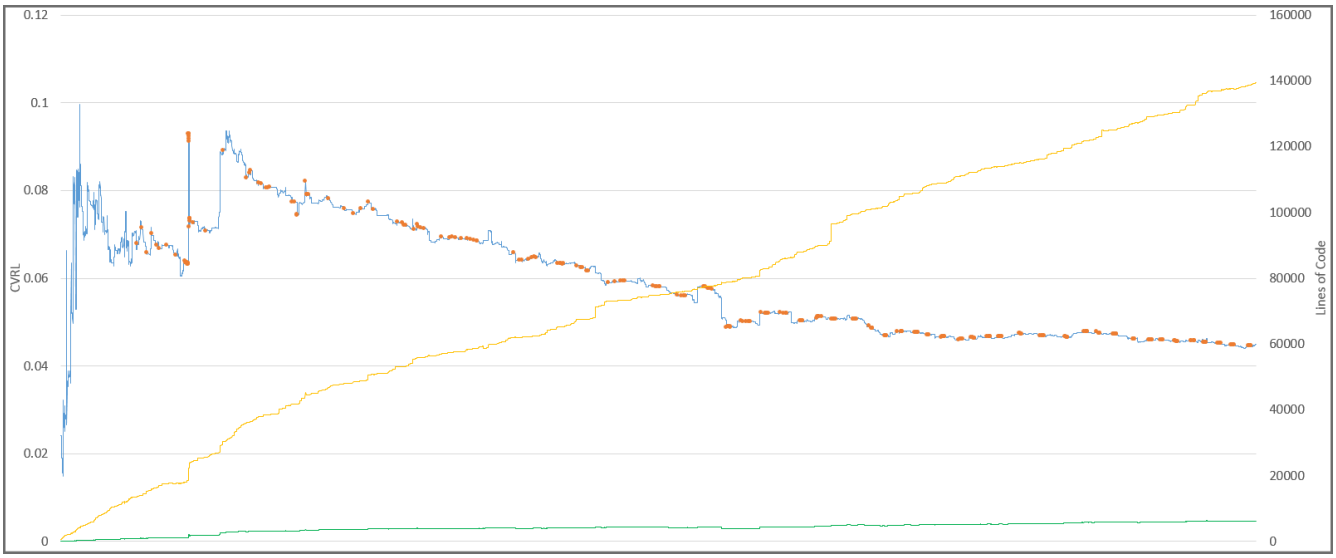

Fig. 3 CVRL, SLOC, and CLOC changes over all commits of Git in chronological order. The blue line represents CVRL, with the orange points along it displaying release points. The yellow line represents SLOC and the green line represents CLOC. The CVRL adheres to the left axis, while the other two metrics adhere to the right axis.

3.3.1 Quantitative Analysis

Git's CVRL graph follows a similar trend as Skynet. 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. Like in Skynet, the initial growth can be attributed to many additions of functionality at the beginning of development. 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 unlike in Skynet's stable period. 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 stability period.

Unlike in Skynet, the first release does not correspond to the beginning of the gradual decrease, but happens during the large growth period. As mentioned about the previous two projects, the CVRL is still significantly smaller than the average CVRL, where even at its peak Git's CVRL is smaller. This may simply be due to good developer habits, such as code being refactored before being committed as we have observed about libcurl. As stated before, we did not do a qualitative analysis on Git, but we reserve the right to look into this in the future. Despite this, the quantitative data still provides us with the ability to characterize the development of Git with the help of the analyses on libcurl and Skynet.

3.4 Implications

Based on the CVRL graphs of each open-source project, it is clear that every project will most likely follow it's own trends. In spite of this, there is a very broad trend which is apparent from the results for each project. All three projects have an initial period of instability and fluctuations, followed by a period of stability. This overall trend suggests a general workflow of development. The beginning of development has a lot of fluctuations because by nature not everything is very concrete and many large design choices are most likely being made or still in discussion. As a result of this, CVRL will change as design choices are made since code will need to be changed or refactored to accommodate for new design choices. Once design has been established, the period of stability begins where additions to the project can be refactored easily beforehand to fit with certain design principles, reducing the code clones added with each commit. While the period of stability may differ depending on what kind of development occurs after the design principles are established, these periods are still mostly stable. For example, libcurl is still developing on functionality, which results in a gradual increase during the period of stability. Meanwhile, Skynet has stopped major development of functionality resulting in an almost completely stable CVRL.

Developers can analyze their own projects in similar fashion in order to discover different ways to improve their development process, such as how the time at which design choices were made affected their software's maintainability. This can be broken down to understanding their development phases and their development habits. The style of development chosen could potentially have a large effect on the trends in clone ratios, as an iterative approach would may contain a lot of instability during sprint periods. If using an iterative approach, this analysis could help identify trends during these instability periods to give insight on how the software is being maintained during such periods and if more refactoring is necessary. On the other hand, if a lot of functionality is developed at a certain stage of a project's lifetime and this project has been difficult to maintain, the clone ratios can illustrate where this maintainability issue originated and enlighten the developer on what kind of functionality to develop initially for future projects to prevent such issues.

The most important developer habit to consider when discussing maintainability is the frequency of refactoring. Looking at the clone ratios over development history can show how often refactoring occurs and whether this frequency is enough to keep the software maintained. Although ideally refactoring occurs before commits and a CVRL graph similar to the stable part of Git's graph is produced, time constraints may not allow for such intense refactoring practices, which is where understanding how frequently refactoring should occur can be vital to a project's maintainability. Finally, to meet the needs of a developer's schedule, past clone metrics can be used to infer exactly what kind of development habits are needed to make the process the most efficient. This can range from how quickly design choices need to be made to finding the right balance between an iterative and non-iterative approach when resources are variable during the development time period.

3.5 Threats to Validity

Since our analysis relies heavily on the output of CCFinderX, it's limitations pose a threat to our data. CCFinderX's inability to detect Type-3 clones could possibly allow for a misinterpretation of the CVRL graph to see more refactoring than actually occurred. This would mostly affect our analysis of Git because we did not have the time to qualitatively analyze it and could not confirm notions on refactoring efforts. The other two projects saw thorough analysis of pivotal commit points which may have had this problem. Despite this, the data we gathered should still hold weight because there were not many refactoring points to consider in the first place, especially in Git. As mentioned earlier carefully analyzing the SLOC changes could still help distinguish these occurrences. Since our data can still potentially distinguish these occurrences and there are a means of confirming such suspicions, CCFinderX's limitations on detecting Type-3 clones should not be an issue. It should be noted that conducting a similar analysis again may benefit from the use of a clone detector which can detect Type-3 clones to mitigate the issue completely.

Although mostly being composed of C/C++ files, each project did contain files of significance to development from different languages, which could mean that development with different languages could be occurring simultaneously, making certain periods more stable because functionality is being added through different undetected means. Since all projects had in fact a majority of C/C++ files, this issue should not be very prevalent as the developers should be more likely to continue development in the language used most frequently for ease of compatibility.

Due to the differing organization structures of each project, we were not able to exclude libraries in our analysis in every project analyzed. Depending on the project, this could drastically change the clone ratios. Whether the libraries should or should not be included are up for debate, as they still do serve a purpose in development, but are not necessarily written by the developer themselves. Along with this, our analysis only cover 3 different projects which in hindsight seem to be very well developed based on their relatively low clone ratios. In order to fully understand the anomaly which is software development, an analysis of code clone ratios over the version evolution of more projects is necessary, in particular projects which have higher clone rates than our three. This will help us distinguish different development patterns as well as understand exactly how libraries affect clone ratios.

4. Conclusion

In this paper, we analyzed code clone ratios over the version evolution of three very different open-source projects. Our analysis primarily focused on the CVRL throughout version evolution in conjunction with the SLOC and CLOC at each particular commit point to understand different parts of the development process. With this data it was possible to determine the role which refactoring played during development, as well as make inferences on possible methodologies of development. Each project displayed very different short term trends, but overall all projects showed a period of instability followed by a period of relative stability which may be attributed to project design not being concrete during the initial phase of development. Our data gives us an understanding of the development process that can also help a developer improve their software during development through similar analysis of their software. Looking at data from such an analysis will highlight instances of large CVRL changes, which can be further examined to understand what these changes meant to the project's development phases and how 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] Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., Roover, C. D. and Inoue, K.: Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity, *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 305–314 (online), DOI: 10.1109/SCAM.2014.17 (2014).

[4] 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).

[5] 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).

[6] Ueda, Y., Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Gemini: Code clone analysis tool, *Proceedings 1st International Symposium on Empirical Software Engineering*, Vol. 2, pp. 31–32 (2002).