An Empirical Study of the Factors Affecting Co-Change Frequency of Cloned Code

Lionel Marks, Ying Zou, Iman Keivanloo

Department of Electrical and Computer Engineering
Queen's University
Kingston, ON
lionel.marks@gmail.com, ying.zou@queensu.ca, iman.keivanloo@queensu.ca

Abstract

Code clones are duplicated code fragments that are copied to re-use functionality and speed up development. However, due to the duplicate nature of code clones, inconsistent updates can lead to defects in software system. We extend the existing studies on the inconsistent co-change characteristics, by investigating further factors that affect clone evolution. We study the effect of development cycles, the number of developers, method names similarity and code complexity. Our empirical study includes six industrial software systems to determine if the observations are statistically significant. We discover that one way to improve maintenance of code clones is to decrease code complexity. We find that increased code complexity leads to a decrease in co-change, which can lead to software defects. Likewise, we find that method name similarity is an important factor on co-change frequency of cloned code. From development cycles point of view, we observe that co-change frequency of cloned code does not change significantly from early to later and from development to defect fixing cycles. As a result, we suggest assigning a higher priority for early refactoring (i.e., within the first six months) of all cloned methods with infrequent co-change focusing on clone classes with low similarity in method names and high code complexity.

1 Introduction

Software maintenance is the act of modifying applications to meet new user requirements and to resolve undesirable behavior (i.e., software defects) in code. It has been estimated that 60% of development effort can be attributed to software maintenance and that accounts for 70% of the cost of a software system [1]. These statistics show that the majority of time and money is spent on software maintenance, making this activity important for investigation.

Code clones are known to cause problems to software maintenance [2]. Cloned code consists of at least two code fragments that are similar. Three levels of clone similarity are described in the literature [3]. Type-1 clones are those exactly the same, line-by-line due to copying and pasting a code fragment. Type-2 clones augment Type-1 clones by allowing variables, types or method names to be different, but the code statements must not have any additions, deletions, or reordering. Type-3 clones are the same as Type-2 clones, but allow additions, deletions, or reordering of code statements. Two fragments sharing similar code are referred to as *clone pair*. In our research, a method is considered a *cloned method* if it forms at least a clone pair with another method in the subject system.

When code clones evolve, the similar code fragments can either be updated together or not [4]. The act of updating similar code together is called a *co-change*. If duplicated code fragments are not updated together, it can lead to defects in the software system [5]; the inconsistent update of clones is measured by co-change frequency. This shows that a considerable portion (e.g., 20% [6]) of the overall code can cause problems due to the possibility of having inconsistent co-change and hinder software maintenance. Therefore it is important to study and identify factors affecting co-change frequency.

The clone ratio has been investigated as a contributing metric for co-change prediction [7]. Other work has determined that a relationship exists between software artifacts and developers [8] and that with more developers, extra communication is required to modify code quickly and without introducing defects. There are some studies (e.g., [9]) on the effects of late propagation (i.e., late re-synchronization of inconsistent clones) on the system fault-proneness. However, in the current state of the art in the software community, there is limited research (e.g., [10] [11] [12]) on investigating why inconsistent co-change occurs, specifically for industrial systems. It is not well studied what conditions in development teams cause the need for more defect fixes in *cloned code*. Without this knowledge, we cannot improve the software maintenance processes causing developers to be unaware of co-changing cloned code. In our research, we study if there are certain factors affecting co-change frequency of clones. For our case study, we analyze 6 different industrial applications.

In summary, we investigate the following research questions:

RQ1: *Are cloned code more defect-prone than non-cloned code?*
In this preliminary research question, we observe that cloned code requires more defect fix effort than non-cloned clone in the subject systems. This observation constitutes our major motivation to explore potential factors contributing to defect-proneness of cloned code.

RQ2: *Does the co-change frequency of clones change through the project lifecycle?*
In general, inconsistency in co-change of cloned code increases defect-proneness [5]. As a potential factor that affects co-change frequency, we study project lifecycles. We study the changes in co-change frequency from early stages (i.e., the first six months) of the software system to later development stages. Furthermore, we examine the changes between defect fixing and development cycles. We observe co-change frequency of clones do not improve or change in later cycles. That is, the *co-change frequency* of early (vs. later cycles) and development stages (vs. defect fixing cycles) can be exploited to predict defect-proneness of the cloned code for future and the overall lifetime of the target code.

RQ3: *Does the code complexity affect the co-change frequency?*
In the following questions, we extend our research by exploring other factors. We observe that cloned methods with more complex code are less likely to co-change frequently. If a developer wishes the cloned method to be co-changed frequently, the method should be given a singular purpose with minimal special cases.

RQ4: *Does developer switch affect the co-change frequency?*
Göde and Harder [13] identified developers as potential factors affecting co-change frequency. We find that the effects of developer switches are application specific. The results of developer switches are dependent upon the cloned methods under investigation.

RQ5: *Do method names associated with cloned code affect the co-change frequency?*
We observe that similar naming of method names can improve co-change. Clones with similar method names co-change significantly more frequently in comparison to clones with less similar names.

The rest of this paper is organized as follows. In Section 2, we summarize the related work. We describe the experimental setup of our study in Section 3 and report our case study results in Section 4. Threats to validity of our work are discussed in Section 5. We conclude and provide insights for future work in Section 6.

2 Related Work

Lague et al. [5] reported that half of the changes to a code clone were co-changed with other instances. Similarly, Krinke [14] observed that half of the changes to clones are inconsistent (i.e., lack of co-change). For cloned code, co-change is important since a lack of co-change in code can lead to defects in software [7]. Therefore, studies have analyzed inconsistent changes in defect fixes (e.g., [15]). Furthermore, mining approaches are proposed to detect inconsistent updates (e.g., [13]).

Using metrics to predict co-change is a relatively new area of study in comparison to metrics to predict defect fixes. Specifically dealing with cloned code, Geiger et al. determined that clone coverage (percent of a file that was cloned) can be used to predict co-change of cloned code [7].

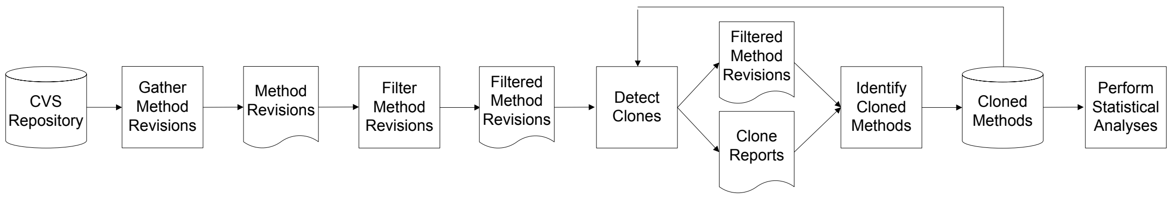

Figure 1 - Major steps of the data collection approach

Kasper et al. [16] analyzed two subsystems for cloned code and found respectively that 60% and 79% of the clones existed in the home directory for each subsystem. The result was explained that it might be harder to find clones when they are not in the home directory (i.e. directory with the majority of clones) of the clone class. Similarly, Thummalapenta et al. [11] studied clone radius, and clone size. Mondal et al. [12] proposed functionality connectivity as a related metric to the co-change problem. Myrizki et al. [10] showed that the size of code and presence of control flow affect the co-change frequency of cloned code.

We continue this work by using different metrics to analyze co-changing cloned code. To find stronger relationships between instances of cloned code, we use the names and signatures of methods inspired by earlier work [17].

We also investigate metrics involving developers and time (i.e., development cycles) due to the research showing links between software artifacts and developers [8]. For example, a study has observed that developers try to minimize the impact of their work on others when making changes to code [18]. Moreover, increased awareness among developers about each other's relevant activities can speed up completion of tasks [8]. Cai and Kim [19] also showed that there exists a relationship between clone survival and such factors (e.g., time and developers). Also, Göde and Harder [13] suggested developers as potential factors affecting co-change frequency. We focus our research on co-change and determine if the number of developers and development cycles significantly impact co-change frequency of cloned code.

3 Case Study Setup

In this section, we introduce the setup of our case study and the implementation for our analysis using clone evolution metrics. We also present the technique used for data cleansing.

3.1 Data Collection

For our case study, we analyze 6 applications. The decision to analyze the 6 applications is to ensure a variety of applications with different functionality are investigated. The subject systems are selected from various domains, e.g., user interfaces, communication protocol, secure transmission, and hardware interfaces. All candidates are medium to large-scale commercial systems written in Java with several years of development. However, due to confidentiality agreements, further information cannot be released about the subject systems.

In summary, Application 1 provides a user interface. Applications 2 and 3 implement protocols for communication. Application 4 is a security application that ensures secure transmission of e-mail. Applications 5 and 6 are used as hardware interfaces.

3.2 Data Processing

The five major steps for extracting co-change activities of clone pairs are shown in Figure 1.

Gathering Method Revisions. The purpose of gathering the method revisions is to provide the necessary data to identify cloned methods. Every change-list in a CVS has a record of the files and lines of code modified from that revision. By synchronizing with the first change-list for a CVS, one can retrieve the code as it was after the very first code revision for a method. Every subsequent change-list can then be used to update the code to its next version. Every time the code of a method was created or updated, we save a copy so that we have all the versions of every method since the code is first created. During this process, a record of all the change-lists for every method revision can be created so that mappings from each method revision can be made to its corresponding change-list. These mappings are later used as a chronological record of the exact time that methods changed together to determine when methods were cloned and for detecting co-change.

Filtering Methods. After gathering all the method revisions, we filter out methods that are not useful for our analysis. Any method that is abstract or a setter/getter was removed from analysis. Abstract methods do not have code within them and therefore cannot contain cloned code. A setter method only has assignment statements (i.e. no control flow, no method calls, etc.). A getter method only returns a value. Although a clone detection tool may find setters or getters to be clones of each other, they are trivial and not useful for analysis of cloned code. Some methods are written for testing the functionality of code. These methods are not included in the final build of code to be shipped to customers; hence excluded from our study.

Detecting Clones. We detect clones in the method revisions by using abstract syntax tree and token based clone detection tools on the method revisions to find Type-1, Type-2, and Type-3 clones. Applications 1-5 have clones identified with SimScan[1], an abstract syntax tree (AST) based clone detection tool. The AST based clone detection tool failed to return a result for Application 6 due to an unexpected error in the SimScan clone detector. To compensate, we use Simian[2], a token based clone detection tool to identify clones in Application 6. All clones used for the study have a minimum of 7 lines of duplicated code.

Incremental clone detection. Since the code under development is constantly changing, we incrementally update the identified cloned methods. An incremental update involves obtaining the new method revisions, filtering the unwanted methods, detecting clones in the new method revisions, merging the clone reports, and identifying the new cloned methods. All of these steps can be done in a day using the original (i.e., non-incremental update) techniques used to identify the cloned methods, except for the step of detecting clones. Clone detection of all the method revisions of an application takes days. To speed up the process for clone detection of an incremental update, we scan the new method revisions with only the most recent existing method revisions.

[1] http://www.blue-edge.bg/download.html
[2] http://www.harukizaemon.com/simian/

Table 1 - Precision of clone detection in each application

App	Clone Detector	Precision
1	SimScan	85.44%
2	SimScan	88.89%
3	SimScan	91.07%
4	SimScan	85.96%
5	SimScan	95.92%
6	Simian	92.31%

Finalizing the clone dataset. The precision of a clone detector is a measure of the percent of relevant clones retrieved. The irrelevant clones (i.e., false positives) are removed from our analysis. We identified the irrelevant clones (i.e., false positives) by consulting with the code owners. From Table 1, we show the precision of clone detection for each application and its clone detector. Applications 1, 2, 4, 5, and 6 have false positive clones that contain many conditional statements. However, the content in the cloned fragments are not similar and are therefore rejected. Application 3 has false positive clones that contain hard coded statements for data output or initialization that are not similar.

The code is analyzed for each application individually because we discovered that each application had its own characteristics: different developers, different code lifetime, etc. This makes analyzing all the applications together less feasible and is consistent with past work from Nagappan et al. [20].

Cloned methods with independent evolutions never co-change. As a result, these methods do not require consistent updates of duplicated code and are removed from analysis of cloned code. We use a minimum number of two co-changes in the cloned methods to ensure a co-changing relationship exists in the methods studied [15].

3.3 Analysis Method

For verification of non-parametric data that has approximately the same shape, a Wilcoxon-Mann-Whitney U test (U test) is appropriate [21]. We use the U test to ensure that our observation is statistically significant. Otherwise the conclusion can be due to the randomness of the studied data. The Wilcoxon-Mann-Whitney U test verifies the data by comparing the medians from the distributions [21]. We perform a two-tailed test to handle statistically significant cases of greater or lesser value than the expected result. For statistical significance, we ensure the finding has a 95% confidence for each application under study. Using a confidence level of 95%, if the p-value is less than 0.05, the data sets are not from the same population and we can reject the null hypothesis under examination. Otherwise, the null hypothesis H_O holds, then the difference is not statistically significant and is likely due to the variability of the data. We use the Wilcoxon-Mann-Whitney U test in all our hypotheses for the case study.

For each experiment, we form our null and alternative hypothesis using the following templates, where y is the experiment number. Each experiment requires two variables that are denoted by α and β.

$$H_{Oy}: \mu(\alpha) - \mu(\beta) \leq 0$$
$$H_{Ay}: \mu(\alpha) - \mu(\beta) > 0$$

We test each hypothesis for all six subject systems independently. If a null hypothesis is rejected by at least 50% (i.e., three systems or more) of our observations, we accept the outcome as a generalizable answer.

4 Case Study Results

This section presents and discusses the results of our five research questions.

RQ1: Are cloned code more defect-prone than non-cloned code?

Motivation. As the preliminary study, we investigate if clones are more defect-prone than non-cloned code in our subject systems. Our work is similar to analysis of defect repositories and fault prediction where correlation or linear models analyses are performed to determine if metrics are relevant to defect fixes [20]. We measure defect-proneness by observing effort related to defect fix activities comparing to other activities. Finally, we investigate if clones are more defect-prone than non-cloned code in our subject systems.

Approach. To represent the defect fix effort, the metric used is the defect fixes per revision shown in Equation (1). The #BugFixes represents the number of defect fixes that are performed on the method under study. The #Revisions represents the number of revisions made to the method.

$$BugFixPerRev = \frac{(\#BugFixes)}{(\#Revisions)} \quad (1)$$

As a result, in this preliminary question, we analyze the features of cloned and non-cloned methods to determine which has a greater percentage of development effort for defect fixes in H_1 and H_2.

H_1 – Cloned code is more defect-prone than non-cloned code. The purpose of H_1 is to determine if the cloned methods have greater problems than non-cloned methods. We formulate the null and alternative hypothesis using Equation (1). In our hypothesis α and β are BugFixPerRev_C and BugFixPerRev_NC respectively where C and NC are referring to cloned and non-cloned methods.

The hypothesis tests the difference in population means between the defect fixes per revision in cloned methods and the defect fixes per revision in non-cloned methods. If H_{O1} is rejected with a high probability (i.e., p-value < 0.05), then we can accept the alternative hypothesis H_{A1}, meaning our suggested hypothesis is proven true. We are then confident that cloned code is more defect-prone than non-cloned code and it requires more defect fix effort.

H_2 – Defect-proneness of the defect-prone cloned methods is greater than defect-prone non-cloned methods. In the previous hypothesis we investigated defect-proneness of any method in the subject system. In this section, we investigate the same problem but only for defect-prone methods. Therefore, in this experiment we exclude any method that is not defect-prone. We use the same statistical non-parametric test to compare the defect fix effort in defect-prone cloned and non-cloned methods. We formulate our hypothesis using the given template. In our hypothesis α and β are BugFixPerRev_BC and BugFixPerRev_BNC respectively where BC and BNC are referring to cloned and non-cloned methods that are defect-prone.

We test the null hypothesis with the difference between the defect fixes per revision in defect-prone cloned methods and the defect fixes per revision in defect-prone non-cloned methods. If the null hypothesis is rejected, we are then confident that the defect-proneness of defect-prone cloned methods is greater than defect-prone non-cloned methods; therefore cloned methods are holding higher priority to be studied in details.

Result. As the first step, we observe whether Mann-Whitney U test is applicable [21]. Figure 2 corresponds to the distribution of defect fixes per revision in cloned and non-cloned methods in all

6 applications. From visual analysis of Figure 2, we note the distributions of cloned and non-cloned methods have the same approximate shape. Large percentages of the methods have defect fixes per revision in the ranges of [0], [0.2, 0.4), and [0.4, 0.6). Since the distributions of the cloned and non-cloned methods have comparatively the same shape, a Mann-Whitney U test is appropriate [21].

1) Defect-proneness of cloned methods vs. non-cloned methods: As the preliminary study, we investigate if clones are more defect-prone than non-cloned code in our subject systems. We considered this research question as a necessary step to our research since our subject systems are holding different characteristics (e.g., development process) than open source systems that are studies before (e.g., [7]).

Of the 6 applications studied in Table 2, the observations made for Applications 1, 2, 3, and 6 are statistically significant and have higher defect fixes per revision in cloned methods. From the fourth column in Table 2, 2 applications are not statistically significant at a confidence level of 95%; applications 4 and 5 have greater defect fix effort in cloned methods. Application 4 has a confidence level of 88% and supports the hypothesis. Application 5 is not statistically significant because 83% and 89% of its cloned and non-cloned methods respectively have zero defect fixes. Given 4 of the applications support the hypothesis with at least a confidence level of 95%, we conclude that H_1 holds.

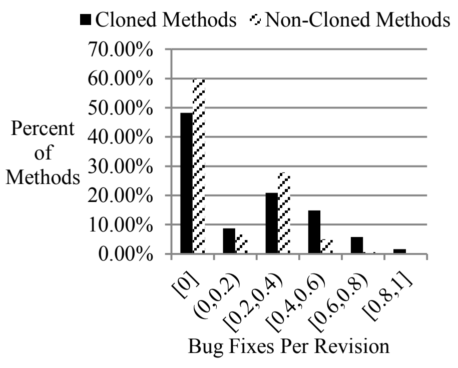

Figure 2 – Distribution of defect fixes per revision for cloned and non-cloned methods in all 6 applications

Table 2 – Comparison of defect fixes per revision for cloned methods and non-cloned methods

App	H[a]	HC[b]	P-Value
1	**0.08**	**0.33**	**0.00**
2	**0.18**	**0.52**	**0.00**
3	**0.08**	**0.53**	**0.02**
4	0.05	0.44	0.12
5	0.01	0.37	0.69
6	**0.28**	**0.61**	**0.00**

a $\mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC)$

b $\frac{\mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC)}{\mu(BugFixPerRev\_C)}$

2) Differences in defect-proneness of defect-prone cloned methods vs. defect-prone non-cloned methods: From H_1, over 48% of the cloned and non-cloned methods do not have defect fixes. Because many of the methods in cloned and non-cloned code do not have any defect fixes, we focus our efforts on the defect-prone methods that have at least one defect fix. Focusing our analysis on the defect-prone methods, we verify if defect-prone cloned methods require greater defect fix effort than defect-prone non-cloned methods to determine which are more problematic.

From Table 3, 5 out of the 6 applications studied are statistically significant with p-values under 0.05 and the defect-prone cloned methods have greater defect fix effort than the defect-prone non-cloned methods. We compare the statistically significant applications with the non-statistically significant application to determine the difference to explain the result. Applications 1, 2, 3, 4, and 6 are statistically significant and have defect-prone cloned methods that duplicate code from relatively new methods. Application 5 does not have a significant difference in the defect fix development effort because 18% of its defect-prone cloned methods duplicate code from older and more tested code (i.e., $\geq$ 6 changes). Less defect fix development effort is needed for methods cloned from the already tested and proven code. As a result, the defect-prone cloned and non-cloned methods for the application are similar in the level of defect fix effort, making the data verification result not statistically significant. Given that 5 of the 6 applications support the hypothesis and are statistically significant, we conclude that H_2 holds.

Table 3 – Comparison of defect fixes per revision for defect-prone cloned and non-cloned methods

App	H[a]	HC[b]	P-Value
1	**0.07**	**0.20**	**0.01**
2	**0.14**	**0.32**	**0.00**
3	**0.07**	**0.21**	**0.02**
4	**0.13**	**0.35**	**0.00**
5	0.01	0.03	0.41
6	**0.18**	**0.36**	**0.00**

a $\mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC)$

b $\frac{\mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC)}{\mu(BugFixPerRev\_BC)}$

Due to the majority of the applications that support the hypothesis, we continue our investigation of the defect-prone cloned methods. Since we observed that clones are more defect-prone, we continue exploring sources that are contributing to defect-proneness rate of clones. If we find these sources, we can exploit them to increase code quality (i.e., via decreasing defect-proneness).

For the rest of the hypotheses in this section, we investigate the reasons affecting co-change frequency of defect-prone methods. Since we are interested in co-change frequency, we study only cloned code within the subject systems.

RQ2: Does co-change frequency change through the project lifecycle?

Motivation. In RQ1, we found that cloned code requires more defect fix effort than non-cloned code. The intuition for a difference in development effort for defect fixes is that if cloned methods are not co-changed, defects can be created [8] [10]. Therefore, in the following research questions, we analyze co-change in defect-prone cloned methods to determine the factors that hinder co-change. Examples of metrics we investigate are code complexity, the number of developers who work on the method, and the similarity of method names. In this section, we focus on effects of project lifecycle on co-change frequency.

Approach. To find common trends in cloned methods for analysis of clone evolution, we analyze the characteristics of each method revision. From these characteristics, we formulate metrics to give us insight into the factors that affect defect fixes and co-change. We introduce the co-change per revision metric to test our hypothesis. The co-change per revision is shown in Equation (2). Empirically, we quantify “co-change frequently” as 75% or more of the revisions in the method are co-changes.

$$CoChangePerRevision = \frac{\#CoChanges}{\#Revisions} \qquad (2)$$

The #CoChanges represents the number of co-changes for the method under investigation. The #Revisions represents the number of method revisions for the method. We use the co-change per revision to divide the defect-prone cloned methods into two data sets for comparison. Defect-prone cloned methods with co-change per revision greater or equal to 75% during development are considered to co-change frequently. Defect-prone cloned methods with co-change per revision less than 75% during development are considered to co-change infrequently.

H_3 – Frequently co-changed defect-prone cloned code in development cycles are also frequently co-changed during defect fixing cycles. To study the co-change of defect fixes in defect-prone cloned methods with frequent co-change and those with infrequent co-change, we present the percent of defect fixes as co-change in Equation (3). The #BugFixesCoChange refers to the number of defect fixes co-changed for the method under investigation. The #BugFixes represents the number of defect fixes for the method.

$$\%BugFixCoChg = \frac{(\#BugFixesCoChange)}{(\#BugFixes)} \qquad (3)$$

Using the metric introduced in Equation (3) and the given template, we formulate our hypothesis. In our hypothesis α and β are $\%BugFixCoChg\_BCF$ and $\%BugFixCoChg\_BCI$ respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively. We test the null hypothesis (i.e. H_{O3}) with the difference in population means between the percent defect fixes as co-changes of the defect-prone cloned methods that co-change frequently and the percent defect fixes as co-change as co-changes of the defect-prone cloned methods that co-change infrequently. If the null hypothesis H_{O3} rejected, we are confident the frequent co-changed defect-prone cloned code in development cycles remain frequent co-changed during the other steps (i.e., defect fixing cycles).

H_4 –Defect-prone cloned methods with frequent co-change in early development have greater co-change than defect-prone cloned methods with infrequent co-change. To continue our study, we investigate if greater co-change in early development leads to greater co-change for the lifetime of the method. However, first, we determine if co-change in early development is significantly different between defect-prone cloned methods with frequent and infrequent co-change. For the analysis of co-change in early development of software, we use the first 6 months to represent early development based on the characteristics of the well-defined software process that is being used for all of the subject systems in our study.

To compare the co-change per revision in the first 6 months in defect-prone cloned methods with frequent and infrequent co-change, we formulate our hypothesis using the given template. In our hypothesis α and β are *CoChgPerRev_First6Mths_BCF* and *CoChgPerRev_First6Mths_BCI* respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively.

The hypotheses test the difference in population means between the co-change per revision in the first 6 months in defect-prone cloned methods with frequent and infrequent co-change. If the null hypothesis H_{O4} is rejected, we are confident the defect-prone cloned methods with frequent co-change have greater co-change per revision in the first 6 months than defect-prone cloned methods with infrequent co-change for the application under study. Hence we can exploit this categorization (i.e., frequent and infrequent co-changed) to distinguish the clones within the first 6 months for our next hypothesis.

H_5 – Infrequently co-changed defect-prone cloned methods in early development cycles become frequently co-changed in later development. This hypothesis focuses only on the defect-prone cloned methods with infrequent co-change. We hypothesize that co-change increases with time because the developers become familiar with the cloned methods in later development. For the analysis of co-change in early and later development of software, we use the first 6 months and after 6 months respectively to remain consistent with our other observations.

We compare the co-change per revision in the first 6 months in defect-prone cloned methods with infrequent co-change to after 6 months defect-prone cloned methods with infrequent co-change. We test the null hypothesis with the difference in population means between the co-change per revision in the first 6 months in defect-prone cloned methods with infrequent co-change and the co-change per revision after 6 months in defect-prone cloned methods with infrequent co-change. If the null hypothesis is rejected, we are confident that the defect-prone cloned methods with infrequent co-change have less co-change per revision in the first 6 months than defect-prone cloned methods with infrequent co-change after 6 months for the application under study. We formulate our hypothesis using the given template. In our hypothesis α and β are *CoChgPerRev_First6Mths_BCI* and *CoChgPerRev_After6Mths_BCI* respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively.

Results. *1) Frequently co-changed defect-prone cloned code in development are also frequently co-changed during defect fixing:* From Table 4, the outcomes of Applications 1, 3, and 5 are statistically significant with p-values under 0.05 and have a greater percentage of defect fixes as co-change for defect-prone cloned methods with frequent co-change. From the developers past behavior of co-changing the cloned methods, defect fixes are also co-changed in the 3 statistically significant applications for the hypothesis. These applications follow the expected trend of co-change during development leads to co-change of defect fixes. Application 2 supports the hypothesis with a confidence level of 79%. Applications 4 and 6 have cases of specialization in the cloned methods that caused our expected trend not to hold. The results in Applications 4 and 6 contradict our hypothesis, with cases respectively of 9% and 19% greater co-change in defect fixes in the specialized methods to fix the coding errors or special cases common to both methods.

Given that 3 of the experiments support the hypothesis with statistical significance, we conclude that H_3 holds. We note that in general, cloned methods that are not specialized (i.e., both methods have clone fragments, but other fragments of the cloned methods are different) benefit from co-change in development and leads to greater co-change in defect fixing cycles.

Table 4 – Comparison of the percent of defect fixes as co-change for defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**0.20**	**0.27**	**0.01**
2	0.12	0.13	0.21
3	**0.44**	**0.47**	**0.04**
4	-0.07	-0.10	0.66
5	**0.48**	**0.72**	**0.05**
6	-0.18	-0.23	0.23

a $\mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI)$

b $\frac{\mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI)}{\mu(\%BugFixCoChg\_BCF)}$

Table 5 – Co-change per revision for first six months of defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**0.18**	**0.29**	**0.00**
2	**0.25**	**0.35**	**0.01**
3	**0.41**	**0.62**	**0.00**
4	-0.08	-0.17	0.77
5	**0.35**	**0.47**	**0.04**
6	**0.39**	**0.56**	**0.01**

a $\mu(CoChgPerRev\_First6Months\_BCF) - \mu(CoChgPerRev\_First6Months\_BCI)$

b $\frac{\mu(CoChgPerRev\_First6Months\_BCF) - \mu(CoChgPerRev\_First6Months\_BCI)}{\mu(CoChangePerRev\_First6Months\_BCF)}$

2) Defect-prone cloned methods with frequent co-change have greater co-change in early development, than defect-prone cloned methods with infrequent co-change: From Table 5, 5 of the 6 applications analyzed have a statistically significant increase in the co-changes per revision in the first 6 months comparing the methods with frequent co-change and the methods with infrequent co-change. The result makes sense that initial co-change of a cloned method maintains the cloned relationship for greater co-change in the future. Application 4 has a confidence level of 23% and showed that the first 6 months did not necessarily lead to greater co-change in the future. From review of the defect-prone cloned methods in Application 4, after the initial cloning, 31% of the methods did not co-change with other defect-prone cloned methods in the first 6 months. However, co-change occurred in later development to handle new updates for the application.

Given that 5 of the 6 applications studied are leading to statistically significant results and support the hypothesis, we conclude that H_4 holds and we can continue our study by focusing on infrequent co-changed clones.

3) Defect-prone cloned methods with infrequent co-change in early development improve with co-change in later development than defect-prone cloned methods with infrequent co-change: From Table 6, Applications 1 and 3 are statistically significant at a confidence level of 95%. Application 3 supports the hypothesis that defect-prone cloned methods with infrequent co-change in the first 6 months can improve in co-change in later development. Application 1 is contrary to the hypothesis and shows more co-change in the first 6 months and in later development co-change decreases. From manual review of Application 3, 75% of the cloned methods that co-changed infrequently are simplified after early development with conditional statements removed. In Application 1, 37% of the cloned methods that co-changed infrequently are updated with 2 to 3 co-changes after cloning in the first 6 months. After the initial development, the functionality for these methods has already been established and only 1 or 2 code modifications are required to maintain the software. As a result, in these methods early development has an average of 50% co-change and later development has a co-change of 0%.

The other 4 applications have a maximum confidence level of 55%. We can explain the lack of statistical significance in the 4 applications. Application 4 has a confidence level of 12%, meaning that the co-change in early development does not change in later development. We attribute this lack of change to the more structured development process of Application 4. We note that Applications 2, 5 and 6 decline in co-change in later development due to increased conditional statements for coding specialization of the methods.

Given that 2 of the applications are statistically significant, with 1 application supporting the hypothesis and the other contrary to the hypothesis, we reject H_5. Based on H_3, H_4, and H_5, we conclude that (1) *frequent co-changed clones* in the first 6 months or development cycles do not become infrequent through project life time and (2) co-change frequency of the *infrequent co-changed clones* do not improve in later development cycles.

Table 6 – Co-change per revision for first six months to after six months of defect-prone cloned methods with infrequent co-change

App	H[a]	HC[b]	P-Value
1	**0.17**	**0.39**	**0.00**
2	0.09	0.20	0.54
3	**-0.40**	**-1.65**	**0.04**
4	0.00	0.00	0.88
5	-0.10	-0.27	0.65
6	-0.11	-0.38	0.70

[a] $\mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI)$

[b] $\frac{\mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI)}{\mu(CoChgPerRev\_First6Mths\_BCI)}$

RQ3: Does the code complexity affect the co-change frequency?

Motivation. To explain the reason that some defect-prone cloned methods co-change more frequently than others, we further investigate code complexity as a potential factor on co-change frequency of cloned methods.

Approach. Since the complexity of a method can affect co-change during development of new features and defect fixes, we use co-change per revision Equation (1) for all code changes to divide the defect-prone cloned methods into methods with frequent co-change and methods with infrequent co-change. We remain consistent with H_3 and use 75% as the threshold between methods with frequent co-change and those that do not.

We use the cyclomatic complexity [22] to represent code complexity. We use cyclomatic complexity since it represents fine grained complexity of the underlying code. To determine whether the additional conditional statements are added to the defect-prone cloned methods that co-change frequently or infrequently, we test the complexity at three different stages of the lifetime of a cloned method: the initial, median, and final revisions.

H_6 – Defect-prone clone classes that co-change frequently have lower code complexity than defect-prone clone classes that co-change infrequently. For the initial, median, and final revisions, we use the statistical non-parametric test to compare the cyclomatic complexity of the defect-prone cloned methods that co-change frequently and defect-prone cloned methods that co-change infrequently. We formulate our hypothesis using the given template. In our hypothesis α and β are Cyclomatic_BCF and Cyclomatic_BCI respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively.

We test H_{O6} with the difference in population means between the cyclomatic complexity in defect-prone cloned methods that co-change frequently and the cyclomatic complexity in defect-prone cloned methods that co-change infrequently. If the null hypothesis H_{O6} is rejected, we are confident the defect-prone cloned methods that co-change frequently have lower cyclomatic complexity than defect-prone cloned methods that co-change infrequently for the application under study.

Result. From Table 7, Applications 1 and 6 are statistically significant and have a lower initial complexity in defect-prone cloned methods with frequent co-change. From Tables 8 and 9, Applications 1, 2, 5, and 6 are statistically significant and have a lower median and final complexity in defect-prone cloned methods with frequent co-change. Applications 2 and 5 have a confidence level of at least 86% in initial cyclomatic complexity, suggesting a reasonable difference exists in the applications between the defect-prone cloned methods that co-change frequently and those that co-change infrequently. Although, we cannot reject the null hypothesis yet; eventually, both become statistically significant as we continue the study (i.e., Table 8 and 9). Applications 3 and 4 are not statistically significant in initial, median, or final cyclomatic complexity. From review of the Applications 3 and 4, 32% of cloned methods with cyclomatic complexity greater than or equal to 7 co-changed frequently. These cloned methods that exhibit frequent co-change are contrary to our hypothesis and explain the lack of statistical significance in Applications 3 and 4. In the statistically significant applications, the methods that co-change less frequently have on average 3.41 more conditional statements in the initial versions. These statements account for special cases specific to the individual cloned method and do not require co-change with other clone methods in the clone class.

The second columns from Tables 7, 8, and 9 show the initial, median, and final values. Four of the statistically significant applications have an increasing trend from initial to final cyclomatic complexity in defect-prone cloned methods that co-change infrequently. The increases in complexity for Applications 1, 2, 5 and 6 from the initial revisions to final revisions, are 2.83, 3.32, 9.99 and 0.66 respectively. An increased value of 1 in cyclomatic complexity refers to one additional conditional statement added from the initial to its final version. As the defect-prone cloned methods that co-change infrequently evolve, more conditional statements are added to handle special cases or different features supported by the application.

Given that 4 of the applications support the hypothesis and have a confidence of at least 86%, we conclude that H_6 holds. From the results of this hypothesis, we note that cloned methods with more complex code are less likely to co-change frequently. If a developer wishes the cloned method to be co-changed frequently, the method should be given a singular purpose with minimal special cases. With less complex code, co-change occurs more frequently due to less specialization of the method.

Table 7 – Comparison of initial cyclomatic complexity of defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**-2.47**	**-0.68**	**0.00**
2	-2.69	-0.69	0.09
3	0.58	0.15	0.44
4	2.63	0.36	0.66
5	-6.04	-0.88	0.14
6	**-2.38**	**-1.36**	**0.01**

[a] $\mu(Cyclomatic\_initial\_BCF) - \mu(Cyclomatic\_initial\_BCI)$
[b] $\frac{\mu(Cyclomatic\_initial\_BCF) - \mu(Cyclomatic\_initial\_BCI)}{\mu(Cyclomatic\_initial\_BCF)}$

Table 8 – Comparison of median cyclomatic complexity of defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**-3.29**	**-0.86**	**0.00**
2	**-4.56**	**-1.12**	**0.00**
3	0.27	0.07	0.56
4	3.10	0.45	0.27
5	**-9.54**	**-1.39**	**0.05**
6	**-3.00**	**-1.50**	**0.03**

[a] $\mu(Cyclomatic\_median\_BCF) - \mu(Cyclomatic\_median\_BCI)$
[b] $\frac{\mu(Cyclomatic\_median\_BCF) - \mu(Cyclomatic\_median\_BCI)}{\mu(Cyclomatic\_median\_BCF)}$

Table 9 – Comparison of final cyclomatic complexity of defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**-5.30**	**-1.43**	**0.00**
2	**-6.01**	**-1.40**	**0.00**
3	0.22	0.05	0.58
4	3.83	0.54	0.13
5	**-16.03**	**-2.44**	**0.03**
6	**-3.04**	**-1.46**	**0.02**

[a] $\mu(Cyclomatic\_final\_BCF) - \mu(Cyclomatic\_final\_BCI)$
[b] $\frac{\mu(Cyclomatic\_final\_BCF) - \mu(Cyclomatic\_final\_BCI)}{\mu(Cyclomatic\_final\_BCF)}$

RQ4. Does developer switch affect co-change frequency?

Motivation. To further clarify the reason that some defect-prone cloned methods co-change more frequently than others, we investigate developer switches in defect-prone cloned methods. We hypothesize that defect-prone cloned methods with frequent co-change have fewer developer switches than defect-prone cloned methods with infrequent co-change.

Approach. The developer switches per revision of the defect-prone cloned methods that co-change frequently are compared with the defect-prone cloned methods that co-change infrequently. To normalize the number of developer switches in relation to the number of method revisions in the method, we calculate the number of developer switches per revision as shown in Equation (4). The #DeveloperSwitches represents the number of developer switches for the method under study. The #Revisions represents the number of method revisions for the method. We subtract 1 from the denominator because a developer switch cannot occur in the first method revision to a method. As a result, the value of the number of developer switches per revision ranges from [0,1].

$$DevSwitchPerRev = \frac{(\#\text{DeveloperSwitches})}{(\#\text{Revisions}-1)} \qquad (4)$$

H_7 – Defect-prone cloned methods with frequent co-change have fewer developer switches than defect-prone cloned methods with infrequent co-change. We calculate the difference in population means between the developer switches per revision in defect-prone cloned methods that co-change frequently and infrequently. We formulate our hypothesis using the given template. In our

hypothesis α and β are DevSwitchPerRev_BCF and DevSwitchPerRev_BCI respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively.

If the null hypothesis H_{O7} is rejected, we are confident the defect-prone cloned methods that co-change frequently have fewer developer switches per revision than defect-prone cloned methods that co-change infrequently for the application under study.

Result. From Table 10, the results of Applications 3 and 6 are statistically significant and have fewer developer switches per revision in defect-prone cloned methods with frequent co-change. From manual review of Applications 3 and 6, 79% of the cloned methods that co-change frequently do not have code changes specific to the method in the conditional statements. The lack of developer switches in these cloned methods limited the number of developers and maintained knowledge of the cloned relationship. Application 2 has a confidence level of 91% and 28% greater developer switches per revision in defect-prone cloned methods with frequent co-change. We investigated the result and the main difference between this application and the others is that these methods implemented protocol requirements. Each new conditional statement provided well-defined functionality. As a result, developer switches did not obstruct co-change of these methods. Applications 1, 4, and 5 are not statistically significant and have a confidence level of less than 43%, meaning that developer switches per revision have little effect on the cloned methods in these applications.

Table 10 – Comparison of developer switches per revision for defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	-0.03	-0.05	0.57
2	0.18	0.28	0.09
3	**-0.28**	**-0.78**	**0.04**
4	-0.06	-0.11	0.75
5	0.10	0.15	0.57
6	**-0.37**	**-1.15**	**0.03**

[a] $\mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI)$
[b] $\frac{\mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI)}{\mu(DevSwitchPerRev\_BCF)}$

Given that only 2 of the experiments are leading to statistically significant result and supporting the hypothesis, we conclude that H_7 is rejected. From the results of this hypothesis, we note that the effects of developer switches are application specific. The results of developer switches are dependent upon the cloned methods under investigation.

RQ5. Do method names affect co-change frequency?

Motivation. Developers give names to methods so that they can refer to them later and understand what the method does. The name of a method implies its functionality. Since cloned methods already represent similar functionality, a similar name for a method implies that there is a greater relationship between them than if the methods do not have similar naming. We hypothesize the relationship is more recognizable and that cloned methods with similar naming co-change more frequently.

Approach. We investigate whether method names in defect-prone cloned methods with frequent co-change are more similar than the method names of defect-prone cloned methods with infrequent co-change. The statistical non-parametric test is used to compare the minimum percent Levenshtein distance in defect-prone cloned methods with frequent co-change and defect-prone cloned methods with infrequent co-change.

H_8 – The method names in defect-prone cloned methods with frequent co-change are more similar than the method names of defect-prone cloned methods with infrequent co-change. We hypothesize the relationship is more recognizable and that cloned methods with similar naming co-change more frequently. We test our hypothesis by analyzing the Levenshtein distance [23] between the cloned method names in their respective clone classes. The Levenshtein distance computes the minimum number of revisions required to change one sequence of characters to another. To make comparisons of method names irrespective of length, we compute the percent Levenshtein distance for each clone pair in a clone class as shown in (5). In the equation, mn_1 and mn_2 refer to two method names.

$$\%LevDist = \frac{LevenshteinDistance(mn_1, mn_2)}{maxLength(mn_1, mn_2)} \quad (5)$$

The LevenshteinDistance(mn_1, mn_2) represents the Levenshtein distance between mn_1 and mn_2. The maxLength(mn_1, mn_2) represents the number of characters from the larger of the two method names mn_1 and mn_2. The percent Levenshtein distance ranges from values [0,1]. Given clone classes can have more than 2 cloned methods; we investigate cases of the most similar naming that exists in a clone class. The most similar naming corresponds to the minimum percent Levenshtein distance between the cloned method under study and the other cloned methods in the clone class.

The null hypothesis H_{O8} is tested with the difference in population means between the minimum percent Levenshtein distance in defect-prone cloned methods with frequent co-change and the minimum percent Levenshtein in defect-prone cloned methods with infrequent co-change. We formulate our hypothesis using the given template. In our hypothesis α and β are Min%LevDist_BCF and Min%LevDist_BCI respectively where BCF and BCI are referring to defect-prone cloned methods with frequent and infrequent co-changes respectively.

If the null hypothesis is rejected, then we are confident the defect-prone cloned methods with frequent co-change have a lower minimum percent Levenshtein than defect-prone cloned methods with infrequent co-change for the application under study.

Result. From Table 11, Applications 1, 3, and 5 are statistically significant with p-values under 0.05 and have lower Levenshtein distances for defect-prone cloned methods with frequent co-change. Application 2, 4, and 6 were not statistically significant and the results are contrary to our hypothesis that similar naming increases co-change.

Table 11 – Comparison of Levenshtein distance for defect-prone cloned methods with frequent and infrequent co-change

App	H[a]	HC[b]	P-Value
1	**-0.11**	**-1.37**	**0.04**
2	-0.04	-0.21	0.55
3	**-0.19**	**-0.78**	**0.01**
4	-0.07	-1.39	0.18
5	**-0.26**	**-0.82**	**0.04**
6	0.11	0.31	0.11

a $\mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI)$

b $\frac{\mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI)}{\mu(Min\%LevDist\_BCF)}$

Manual inspection of the method names in Application 2 revealed that 83% of cloned methods in different classes with the exact same name co-changed infrequently. Application 4 showed 31% less similar naming in the defect-prone cloned methods that co-changed frequently. In Application 6, 50% of the cloned methods with frequent co-change have substring matches in the method names (i.e. setActive() and setInactive()). Given that 3 of the results of applications are statistically significant and support the hypothesis, we can conclude that H_8 holds. Similar naming of method names can improve co-change.

5 Threats to Validity

We now discuss the different types of threats that may affect the validity of the results of our experiment.

External validity tackles the issues related to the generalization of the result. Our study performs analysis on 6 applications. The 6 applications represent a variety of types of applications. Each of the 6 applications has a large number of cloned and non-cloned methods. Nevertheless, we should, in the future, test our hypotheses using other applications to determine if our results hold for other software systems (e.g., open source).

In our hypotheses, we use the defect-prone cloned and non-cloned methods for analysis of defect fixes. A defect-prone method is defined as a method with at least one defect fix. We specifically analyze only methods with defect fixes because our study shows that the non-defect-prone methods are not as significant. In the future, we can extend the analysis of defect fixes to include methods without defect fixes to establish more general results on cloned non-cloned methods.

Internal validity is a concern with the issues related to the design of our experiment. The use of two different clone detectors can potentially introduce a bias in the study. Our AST-based clone detector can identify less similar duplicated code fragments (i.e., Type-3) than our token-based clone detector. We mitigate the discrepancy through manual review of the cloned methods to ensure each cloned method analyzed is appropriate for clone evolution study.

In the applications under study, a mapping between a defect repository and the CVS change-lists was not available. As a result, to categorize a change-list as a defect fix, we search the change-lists for words that involve defect fixes (i.e. bug, fix, etc.). We perform manual review of each change-list to ensure the only methods updated are associated with the defect fix. As a result of the manual review, we are certain no false positives occurred in our study. However, false negatives may occur due to developers that omitted mention of the defect fix in the change-list description or because the test suites missed defects that exist in the software.

Defect fixes in cloned methods are counted on the method level because we consider a method to be a reasonable unit of work that can be assigned to a person. As a result, we do not differentiate if defect fixes in a cloned method are in the cloned or non-cloned code. The number of defect fixes reported in cloned methods may not reflect solely the number of defect fixes in cloned code.

Methods with more method revisions have more chances to change in complexity and have more defect fixes. For the metrics in our study, we use normalization techniques to make comparisons that take into account the numbers of method revisions or developers in each method studied. Our normalization techniques are bias-free of all metrics except for code complexity. We analyze the change in code complexity in software evolution by comparing the complexity of the initial, median, and final versions of each method. The technique is useful for obtaining data on the changes in complexity throughout the lifetime of the method. However, methods with a greater number of method revisions can be compared against methods with fewer method revisions in the median and final versions of the methods. For example, if we consider methods A and B. Method A has 5 method revisions. Method B has 13 method revisions. In a comparison of the complexity of the final versions of the methods, we compare the complexity of a method with only 5 method revisions, to a method with 13 method revisions. Although the number of method revisions does not necessarily increase the complexity, the analysis can be biased due to a larger number of method revisions in Method B. One possible solution is to use the 1st, 5th, and 10th method revisions of each method. However, methods with fewer than 10 method revisions are excluded from the analysis of the 10th method revision. Methods with greater than 10 method revisions have complexity data that is excluded from analysis. We currently do not have a solution to handle the problem and suggest further research to create a better normalized approach to the investigation of changes in complexity in software evolution. To measure complexity, we chose to use cyclomatic complexity even though lines of code have been found to be as accurate in the prediction of defect fixes [24]. Cyclomatic complexity counts the number of decisions in the code as opposed to treating each line of code as equally complex. By tracking the number of decisions, we can note if new logical branches are added or removed during development.

Construct validity is a concern as to the meaningfulness of the measurement. In our study, the results from our analyses are application specific. The characteristics of each application provide different reasons to explain the development effort of defect fixes in cloned and non-cloned methods. The coding styles used for the cloned methods influenced our results. Some coding styles inherently allow for a greater or fewer numbers of co-changes with respect to the total number of method revisions to the method. As a result, the applications and number of applications that support our hypotheses differs between experiments.

6 Conclusion

In this work, we study the effect of development cycles, the number of developers, method names similarity and code complexity on co-change frequency of clones. Our empirical study includes six industrial software systems. We discover that one way to improve maintenance of code clones is to decrease the code complexity. We find that increased code complexity leads to a decrease in co-change, which can lead to defects in the software. Likewise, we find that method name similarity is an important factor on co-change frequency of cloned code. Moreover, based on our research, if the cloned method does not co-change well in the first 6 months, the cloned method is unlikely to co-change well in the future.

Using techniques to refactor clones, the clones with low co-change can be refactored and the overall co-change of the cloned methods in the application can be increased. As a result, we suggest assigning a higher priority for early refactoring (i.e., within the first six months) of all the cloned methods with infrequent co-change focusing on clone classes with low similarity in method names and high code complexity. In the future, we plan to analyze more applications from different sources, such as open source projects to verify our findings from our current industrial applications. Then, based on the agreement between their analyses we can get more fair evaluation of factors that influence defect fixes and co-change. We plan to consider additional measurement approaches such as including the longest common subsequence for the name similarity factor.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. Communications of the ACM, June 1978.

[2] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. International Conference on Software Maintenance, pp. 227-236, 2008.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. IEEE Trans. Software Engineering, September 2007.

[4] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. Working Conference on Mining Software Repositories, 2013.

[5] B. Lague, D. Proulx, J. Mayrand, E. Merlo and J. Hudeophl. Assessing the benefits of incorporating function clone detection in a development process. International Conference on Software Maintenance, 1997.

[6] G. Antoniol, E.Merlo, U. Villano and M. D. Penta. Analyzing cloning evolution in the Linux kernel. Information and Software Technology, 2002.

[7] R. Geiger, B. Fluri, H. C. Gall and M. Pinzger. Relation of code clones and change couplings. International Conference of Fundamental Approaches to Software Engineering, 2006.

[8] C. Gutwin and S. Greenberg. The effects of workspace awareness support on the usability of real-time distributed groupware. Transactions on Computer-Human Interaction, September 1999.

[9] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. International Conference on Software Maintenance, 2011.

[10] M. S. Yudha, R. Asano, H. Aman. A feature analysis of co-changed code clone by using clone metrics. Transactions on Fundamentals of Electronics IEICE, Communications and Computer Sciences, 2012.

[11] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. Empirical Software Engineering, February 2010.

[12] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of co-changed method groups: a case study on open source systems. Conference of the Center for Advanced Studies on Collaborative Research, 2012.

[13] N. Göde and J. Harder. Oops! . . . I changed it again. International Workshop on Software Clones, 2011.

[14] J. Krinke. A study of consistent and inconsistent changes to code clones. Working Conference on Reverse Engineering, 2007.

[15] L. Aversano, L. Cerulo and M. D. Penta. How clones are maintained: an empirical study. European Conference on Software Maintenance and Engineering, 2007.

[16] C. Kasper and M. Godfrey. Aiding comprehension of cloning through categorization. Workshop on Principles of Software Evolution, 2004.

[17] I. Neamtiu, J. Foser and M. Hicks. Understanding source code evolution using abstract syntax tree matching. Mining Software Repositories, 2005.

[18] C. Souza and D. Redmiles. An empirical study of software developers' management of dependencies and changes. International Conference on Software Engineering, 2008.

[19] D. Cai and M. Kim. An empirical study of long-lived code clones. International Conference on Fundamental Approaches to Software Engineering, 2011.

[20] N. Nagappan, T. Ball and A. Zeller. Mining metrics to predict component failures. International Conference on Software Engineering. 2006.

[21] W. Hays. Statistics. New York: Harcourt Brace College Publishers, 1994.

[22] T. McCabe. A complexity measure. IEEE Transactions on Software Engineering, December 1976.

[23] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics, pp. 707-710, February 1966.

[24] T. L. Graves, A. F. Karr, J.S. Marron and H. Siy. Predicting fault incidence using software change history. IEEE Transactions on Software Engineering, July 2000.