An Insight into the Dispersion of Changes in Cloned and Non-cloned Code: A Genealogy Based Empirical Study

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider

University of Saskatchewan, Canada

{*mshankar.mondal, chanchal.roy, kevin.schneider*} *@usask.ca*

Abstract

In this paper, we present an in-depth empirical study of a new metric, *change dispersion*, that measures the extent changes are scattered throughout the code of a software system. Intuitively, highly dispersed changes, the changes that are scattered throughout many software entities (such as files, classes, methods, and variables), should require more maintenance effort than the changes that only affect a few entities. In our research we investigate change dispersion on the code-base of a number of subject systems as a whole, and separately on each system's cloned and non-cloned code. Our central objective is to determine whether cloned code negatively affects software evolution and maintenance. The granularity of our focus is at the method level.

Our experimental results on 16 open-source subject systems written in four different programming languages (Java, C, C#, and Python) involving two clone detection tools (CCFinderX and NiCad) and considering three major types of clones (Type 1: exact, Type 2: dissimilar naming, and Type 3: some dissimilar code) suggests that change dispersion has a positive and statistically significant correlation with the change-proneness (or instability) of source code. Cloned code, especially in Java and C systems, often exhibits a higher change dispersion than non-cloned code. Also, changes to Type 3 clones are more dispersed compared to changes to Type 1 and Type 2 clones. According to our analysis, a primary cause of high change dispersion in cloned code is that clones from the same clone class often require corresponding changes to ensure they remain consistent.

Keywords: Dispersion, Instability, Clones, Software Maintenance, Code Change

1. Introduction

Code cloning is a common yet controversial practice that studies have shown to have both positive [1, 5, 9, 11, 14, 15, 7] and negative [10, 16, 17, 18] implications during software development and maintenance. Reuse of code fragments with or without modifications by copying and pasting from one location to another is very common in software development. This results in the existence of the same or similar code blocks in different components of a software system. Code fragments that are exactly the same or are very similar to each other are known as clones. Three types of clones are commonly studied: Type-1 (exact clones), Type-2 (clones with dissimilar naming), and Type-3 (clones with dissimilar naming and/or with some dissimilar code). The impact of clones on software maintenance is of great interest. Researchers have investigated the stability of cloned and non-cloned code [5, 13, 14, 15, 16, 17, 22, 19, 21, 6] using a number of approaches in order to quantify clone impact. The idea is that if cloned code is less stable (e.g., numerous changes) than non-cloned code during maintenance, it is an indication that clones require more maintenance effort than non-cloned code [5]. If this is the case, clones may be considered harmful in the maintenance phase. However, existing approaches for measuring stability are insufficient.

Most of the stability measurement methods [5, 9, 14, 17] calculate stability in terms of code change, however one method [15] calculates stability in terms of code age. The following three main approaches have been used to measure stability: (i) calculate the ratio of the total number of lines added, deleted and modified in a code region to the total number of lines in the code region; (ii) determine the modification frequency of a code region where the modification frequency considers the number of occurrences of consecutive lines added, deleted or modified [9]; and (iii) calculate the average last change dates of cloned and non-cloned code regions using SVN's blame command [15].

1.1. Motivation

The existing stability measurement approaches fail to investigate the following important aspect regarding change.

When comparing the stability of two code regions, it is also important to investigate how many different entities in these two code regions have been affected (i.e., changed).

Explanation: We consider two code regions, *Region 1* and *Region 2*, in a software system and each of these two regions has the same number

of program entities (suppose 100 entities). If during a particular period of evolution of a software system, the changes that occurred in *Region 1* affected 10 entities while the changes in *Region 2* affected 50 entities, this phenomenon has the following implications.

- **Implication 1 (Regarding change-proneness):** The program entities in *Region 2* are more change-prone than the program entities in *Region 1* for the particular period of evolution regardless of the number of changes that occurred in each of these regions.

- **Implication 2 (Regarding change effort):** The amount of uncertainty in the change process [8] in Region 2 is higher compared to the uncertainty in the change process in Region 1 because, on the basis of the change-proneness of entities during this particular period, while each of the 50 entities in *Region 2* has a probability of getting changed in near future (possibly, in the next commit operation), only 10 entities in *Region 1* have probabilities of getting changed. It is also likely that the requirement specifications corresponding to higher number of entities in *Region 2* are more unstable compared to *Region 1.* Thus, the entities in *Region 2* are likely to require more change effort compared to the entities in *Region 1.*

  This is also possible that the entities in *Region 2* are more coupled than the entities in *Region 1.* In other words, changes in one entity in *Region 2* possibly require corresponding changes to a higher number of other entities compared to *Region 1.* Higher coupling among program entities might cause ripple changes to the entities and thus, can introduce higher change complexity as well as effort[1].

  Thus, if it is observed that during the evolution of a software system, higher proportion of entities in the cloned region were changed compared to the proportion of entities changed in non-cloned region, then it is likely that cloned region required higher change effort than non-cloned region for that subject system.

Considering the above two implications regarding stability we introduce a new measurement metric: *change dispersion.* We calculate *change dispersion*

[1]Coupling among Entities: http://www.avionyx.com/publications/e-newsletter/issue-3/126-demystifyingsoftware-coupling-in-embedded-systems.html

Table 1: Research Questions

Serial	Research Question (RQ)
Research Questions Regarding Change Dispersion in Cloned and Non-cloned Code	
RQ 1.	Is the change dispersion in cloned code higher than the change dispersion in non-cloned code?
RQ 2.	Do different types of clones exhibit different change dispersion? If so, which type(s) of clones shows higher change dispersion compared to the others?
RQ 3.	Do cloned and non-cloned code in the subject systems of different programming languages show different comparative scenarios of change dispersion?
Research Questions Regarding Change Dispersion and its Relation with Source Code Change-proneness (or Instability)	
RQ 4.	Is higher dispersion of changes an indicator of higher instability (or change-proneness) in the source code?
RQ 5.	If cloned code has higher change dispersion than non-cloned code, then does it also indicate that cloned code is more change-prone (or unstable) than non-cloned code?
RQ 6.	How does change dispersion in clones affect the stability of cloned code?

using method level granularity. The definition of change dispersion with related terminology will be presented in the next section.

1.2. Objective

We perform an in-depth investigation of change dispersion with the central objective of gaining insight into the relative change-proneness (i.e., instability) of cloned and non-cloned code during software maintenance, and investigating ways to minimize the change-proneness of clones. Intuitively, high change-proneness may indicate high maintenance effort and cost. Also, frequent changes to a program entity has the potential to introduce inconsistency in related entities. As such, change-proneness may have important implications for software maintenance. Existing studies regarding clone impact have resulted in controversial outcomes using a variety of different metrics. Our proposed metric, change dispersion, measures an important characteristic of change that has not been investigated before. We perform a fine-grained analysis of clone impact using change dispersion and answer six research questions presented in Table 1.

1.3. Findings

On the basis of our experimental results on 16 subject systems covering four different programming languages (Java, C, C#, and Python) considering three major types of clones (Type-1, Type-2, and Type-3) involving two clone detection tools (CCFinderX[2] and NiCad [24]) we answer the six research questions. The answers (elaborated in Table 9) can be summarized as follows.

Change dispersion has a positive, and statistically significant correlation with source code instability (i.e., change-proneness). Higher change dispersion in the code-base can also be an indicator of higher coupling among program entities. Cloned code, especially in the subject systems written in Java and C, often exhibits higher change dispersion and is possibly more change-prone compared to non-cloned code. Moreover, Type 3 clones have the tendency of getting more dispersed changes compared to the other two clone types (Type 1 and 2). According to our analysis, a primary reason behind higher change dispersion in cloned code is that clone fragments from the same clone class often require corresponding changes in order to ensure they remain consistent.

Clone refactoring might be helpful to reduce change dispersion as well as change-proneness in cloned code and can potentially help decrease clone maintenance. However, some clones may implement cross cutting concerns [2] and refactoring these clones might not be beneficial [4]. The challenge of clone maintenance suggests it would be beneficial to have an automated system to facilitate clone management.

1.4. Paper Organization

The rest of the paper is organized as follows. Section II describes related terminology. Section III elaborates on change dispersion, and Section IV describes the steps in calculating change dispersion. Section V contains the experimental setup and the experimental results are presented in Section VI. Section VII describes possible threats to validity. Section VIII outlines the relevant research, and Section IX contains concluding remarks and future work. This paper is an extended version of our earlier work [20].

[2]CCFinderX: `http://www.ccfinder.net/ccfinderxos.html`

2. Terminology

This section defines key terminology used in the paper to describe method-level software change. In subsequent sections we introduce a number of metrics, and in those sections we will define specific terms relevant to the metric definitions.

2.1. Change

We use the same definition of *change* as that defined by Hotta et al. [9]. According to their definition, a single change can affect multiple consecutive lines. Suppose n lines of a method have been changed through additions, deletions or modifications. If the n lines lines are consecutive, then the number of changes is one. If the n lines are not consecutive, then the number of changes is equal to the number of unchanged portions within the n lines plus one.

2.2. Dispersion of Changes

Dispersion of changes in a code region (cloned, non-cloned or the entire code-base) refers to the percentage of methods affected by changes in that region during a period of evolution. For each system we studied, the evolution period ranged from the first to the last revision we collected for that system. Table 2 indicates the number of revisions we considered for the software systems we studied. All of the commit operations during the evolution period that involved some modifications to the source code are taken into account.

Change dispersion is related to *change entropy*, introduced by Hassan [8] (based on Shannon Entropy [30]) to quantify the uncertainty of changing program artifacts. Change entropy of a system for a particular period of time depends on the distribution of changes over the entities in the system for that time period. However, change dispersion during a particular period is different in the sense that it depends only on the number of entities changed in that period. The following two points will clarify this.

Point 1. *For different distribution of changes to the same number of entities in two different periods, the change entropies will be different. However, the change dispersions for these periods will be the same.*

Explanation. Let us assume that a system has 4 entities: *e1*, *e2*, *e3*, and *e4*. During a particular period of time only 2 entities,*e1* and *e4*, changed. Then, change dispersion of this system for this time period = 2*100 / 4 = 50 (i.e., 50% of the entities were changed) regardless of which entity changed

how many times. However, change entropy of this system for this time period depends on the distribution of changes to *e1* and *e4*. Suppose, *e1* changed 5 times and *e4* changed 10 times during this period. Then, change probability of $e1 = 5/(5+10) = 0.33$ and change probability of $e4 = 10/15 = 0.66$. Thus, change entropy $= -(0.33 * log_2(0.33) + 0.66 * log_2(0.66)) = 0.92$ according to the equation [8] for calculating change entropy. If *e1* changed 6 times and *e4* changed 11 times, then the change probabilities of *e1* and *e4* are 6/17 (= 0.35) and 11/17 (= 0.65) respectively. At this time, change entropy $= -(0.35 * log_2(0.35) + 0.65 * log_2(0.65)) = 0.93$, however, change dispersion = 50 as only two entities (i.e., 50% of the entities) changed. In this way, for different distribution of the number of changes to *e1* and *e4*, the change entropy will be different.

Point 2. *This is mathematically possible that for two different number of changed entities during two different periods, the change entropies will be the same because of the distribution of changes to the entities. However, the change dispersions for these periods will be different.*

Explanation. We consider the above example of four entities. Let us assume two periods, *Period 1* and *Period 2*. In *Period 1*, two entities, *e1* and *e2*, were changed. Each of *e1* and *e2* changed 5 times (10 times in total). Then, the change entropy for *Period 1* is 1 (according to the equation [8]). Suppose, in *Period 2*, three entities *e1*, *e2*, and *e3* were changed. *e1* changed 400 times, *e2* changed 596 times and, *e3* changed 4 times. The change entropy for this distribution (in *Period 2*) is also 1. Thus, different number of entities might change in two different periods, but their entropies can be the same. However, for the above two periods (*Period 1* and *Period 2*), the change dispersions are, 50 (2*100/4 = 50% of the entities changed in *Period 1*) and 75 (3*100/4 = 75% of the entities changed in *Period 2*) respectively.

From the above two points we see that while change entropy solely depends on the distribution of changes to the entities, change dispersion only depends on the number of changed entities. As our main focus is on the number of entities changed in the system during a particular time period and not on the distribution of the changes to the entities during that period, we consider change dispersion to be the appropriate measure for our purpose.

2.3. Method Genealogy

During the evolution of a software system a particular method might be created in a particular revision and can remain alive in multiple consecutive

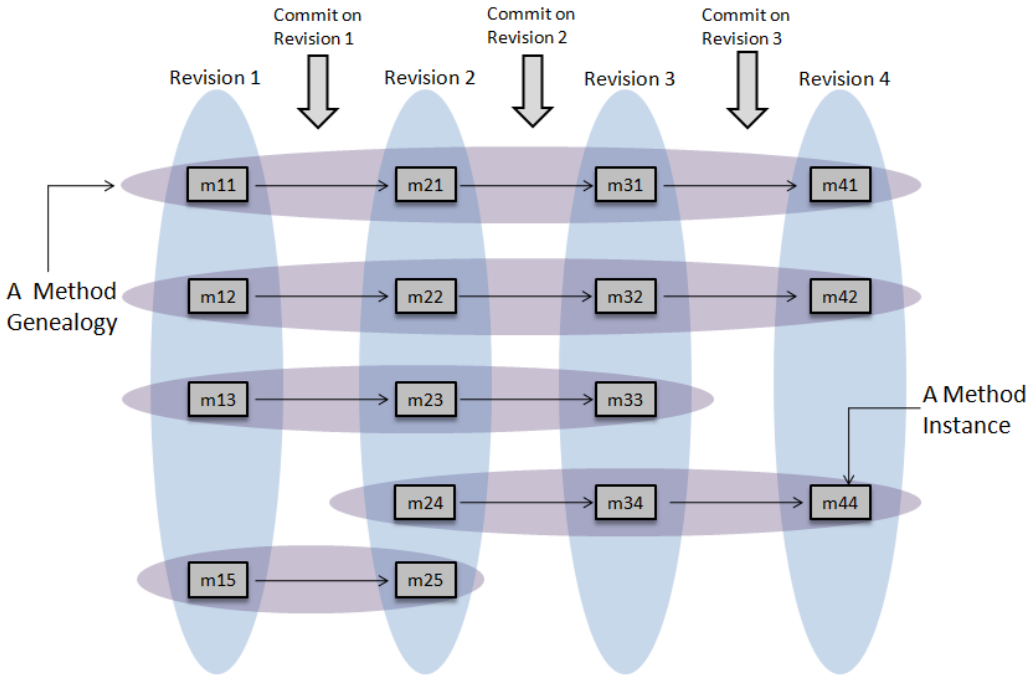

Figure 1: Examples of Method Genealogies

revisions. Each of these revisions has a separate instance of this method. Method genealogy considers that all of these method instances belong to the same method. By inspecting the genealogy of a particular method we can determine whether the method was changed during maintenance. It is possible that, a particular method will remain unchanged during the maintenance phase. In that case, all the instances of this method (in different revisions) will be the same.

In Fig. 1 we show five examples of method genealogies. We see that there are four revisions in total. A commit operation for a particular revision creates the immediately next revision. An example method genealogy (the top most one) in this figure consists of the method instances: m11, m21, m31, and m41. These are four instances of the same method. A commit operation applied to a revision might not change all the method instances in that revision. If the commit operation on 'Revision 1' makes changes to the method instance m11, m21 will be different than m11. Otherwise, m11 and m21 will be the same.

Detecting Method Genealogies: Lozano and Wermelinger [18] proposed an approach for detecting method genealogies. We followed their approach in our study. There are basically two steps in detecting method genealogies from a given set of revisions for a system. These are:

- **Method Detection:** Detecting methods for each of the given revisions; and,
- **Method Mapping:** Making a one-to-one correspondence between the methods in every two consecutive revisions.

For detecting methods we used CTAGS[3]. Methods are detected along with their signatures and location information. The location information consists of the file, package (in case of Java), and class (in case of Java and C#) in which the method exists. After detecting the methods in all revisions we perform method mapping. Method mapping is performed in the following way. Suppose, m_i is a method in revision R_i. In order to find the corresponding instance of this method in revision R_{i+1} we consider two cases.

Case 1: If a method m_{i+1} in R_{i+1} has the same signature and location information as m_i, m_{i+1} is an instance of m_i. The content of these two methods might be the same or different. If the commit operation applied to R_i makes some changes to the method instance m_i, the contents of m_i and m_{i+1} will be different.

Case 2: For m_i in R_i we might not locate a method in R_{i+1} with the same signature and location information. In that case, we detect two sets of methods in R_{i+1}. The first set S_{s-l} contains those methods that have the same signature but different location, and the second set S_{l-s} contains those methods that have the same location but different signatures. The methods in these two sets are called candidate methods. We then compute the similarity between m_i and each of the candidate methods in the first set using the Strike A Match algorithm[4] (an algorithm for determining text similarity) and record the best similarity value and the corresponding candidate method. If this value is above 70% we consider the associated candidate method as the instance of m_i in R_{i+1}. If no candidate method in the first set has a similarity

[3]CTAGS: http://sourceforge.net/projects/ctags/develop?source=navbar

[4]Strike A Match algorithm: `http://www.catalysoft.com/articles/StrikeAMatch.html`.

value greater than 70%, we go through the same process with the second set. If no method in the second set has a similarity value greater than 70% then, m_i is considered to have been deleted in revision R_{i+1}.

After performing the mapping operation between every two consecutive revisions we obtain the method genealogies (Fig. 1).

2.4. Instability

We use the term *instability* to refer to the source code change-proneness of a subject system. Change-proneness is based on how often the source code of a software system changes, typically in terms of changes to a program entity, such as a method. A number of aspects can be taken into consideration when determining change-proneness (i.e., instability), including the size of the entity and change over time. In Section 6.5 we define metrics for measuring the instability of methods, and a detailed description of the metrics is described there.

2.5. Consistency Ensuring Changes to Clones

A clone class is the set of all code fragments that are clones of one another. If two or more clone fragments from the same clone class change together, because changes in one clone fragment required corresponding changes to other clone fragments, then these changes are considered to be *consistency ensuring changes to clones*. In our study we use two clone detection tools, NiCad and CCFinderX. NiCad provides us with clone detection results that separates clones into different clone classes, and so we can use NiCad results directly to determine *consistency ensuring changes to clones*. CCFinderX, however, provides us only with clone pairs and so we must first process CCFinderX results to obtain clone classes.

3. Calculation of Change Dispersion

We calculate change dispersion on the basis of the changes to methods. A method is defined as a cloned method when it contains some cloned lines. Based on this definition, there are two types of cloned methods: (i) fully cloned methods (all of the lines contained in these methods are cloned lines) and (ii) partially cloned methods (these methods contain some non-cloned portions). For calculating the dispersion of changes in cloned code, we consider changes in the cloned portions of the fully or partially cloned methods. Partially cloned methods are also considered when calculating the dispersion

of non-cloned code, because changes might occur in the non-cloned portions of the partially cloned methods. Also, while determining method genealogies it might be the case that a partially cloned method has become fully cloned or fully non-cloned after a change. These methods are considered when calculating the dispersions of both cloned and non-cloned code. Here, we should mention that we have not considered changes to the preprocessor directives while calculating change dispersion. As we conduct our experiment considering method level granularity and preprocessor directives are not common in method-bodies, changes to the preprocessor directives do not have any considerable impact on our experiment results.

Given a subject system with R revisions, we first find the methods and their boundaries in each revision and then extract the method genealogies. We also use a clone detection tool on each revision to determine the clone blocks. With the clone block and method information, we determine which methods contain clone blocks and count the number of method genealogies with cloned code and the number of method genealogies with non-cloned code. We also determine the changes between consecutive revisions and if the changes affect the cloned or non-cloned portions of the methods.

Suppose, for a subject system, that the number of method genealogies with cloned code is C and the number of method genealogies with non-cloned code is N. During the evolution of the software system, the number of method genealogies with cloned code that receive some changes to their cloned portions is C_c. Generally, C is greater than C_c. Also, N_c is the number of method genealogies that have some changes to their non-cloned portions.

Change Dispersion in Cloned Code (CD_c): We calculate the dispersion of changes in cloned code (CD_c) according to the following equation.

$$CD_c = \frac{C_c \times 100}{C} \tag{1}$$

Change Dispersion in Non-cloned Code (CD_n): The dispersion of changes in non-cloned code (CD_n) is calculated using the following equation.

$$CD_n = \frac{N_c \times 100}{N} \tag{2}$$

Change Dispersion in the Entire Code-base (*CDEC*): We also wanted to determine whether higher dispersion of changes to methods in the entire source code is an indicator of higher instability (i.e., change-proneness) in the source code. If a subject system has G method genealogies (cloned or

non-cloned) in total and G_c of these genealogies received some changes during the evolution, the dispersion of changes to the methods of this subject system ($CDEC$) can be calculated by the following equation.

$$CDEC = \frac{G_c \times 100}{G} \tag{3}$$

The total number of method genealogies in a subject system (G) might be smaller than the summation of the counts of method genealogies selected separately for cloned (C) and non-cloned code (N), because as mentioned in our discussion (at the very beginning of this section) some method genealogies (the partially cloned method genealogies) can be considered for calculating both CD_c and CD_n.

4. Steps for Calculating Change Dispersion

The following subsections describe how we obtained the subject systems and how we calculated change dispersions for cloned and non-cloned code.

Extraction of Repositories: All of the subject systems (listed in Table II) on which we have applied our method to calculate change dispersion were downloaded from open source SVN repositories. For a subject system, we extracted only those revisions which were created because of some source code changes (additions, deletions or modifications). To determine whether a revision should be extracted or not, we checked the extensions of the files which were modified to create the revision. If some of these modified files are source files, we considered the revision to be a target revision.

Preprocessing: We applied two preprocessing steps on the source files of each target revision of the subject systems written in Java, C and C# before clone detection. They are: **(i)** the deletion of lines containing only a single brace ('{' or '}') and appending the brace at the end of the previous line and **(ii)** the removal of comments and blank lines. For Python systems we removed the comments and the associated blank lines created due to the removal of comments.

Method Detection and Extraction: For detecting the methods we applied CTAGS[2] on the source files of a revision. For each method we collected: (i) package name (Java), (ii) file name, (iii) class name (Java and C# systems), (iv) method name, (v) signature, (vi) starting line number and (vii) ending line number. We also assigned a unique id to each method within each revision. However, the id of a method of one revision can be

the same as that of a method of another revision. This does not introduce conflicts because a separate file is generated for each revision.

Clone Detection: We used CCFinderX[1] and NiCad [24] to detect clones in our experiment. CCFinderX is a token based clone detection tool that currently detects block clones of Type-1 and Type-2. Also, NiCad is a recently introduced clone detection tool that can detect three types of clones (Type-1, Type-2, and Type-3) with high precision and recall [23] considering both block level and method level granularities.

We applied the two clone detection tools to each target revision to detect clone blocks. The clone blocks were then mapped to the already detected methods of the revision by comparing the beginning and ending line numbers of clone blocks and methods. For each method we collect the beginning and ending cloned line numbers (if they exist). CCFinderX currently outputs the beginning and ending token numbers of clone blocks. We automatically retrieve the corresponding line numbers from the generated preprocessed files.

Detection and Reflection of Changes: We identified the changes between corresponding files of consecutive revisions using the UNIX *diff* command. *diff* outputs three types of changes with corresponding line numbers: (i) addition, (ii) deletion, and (iii) modification. We mapped these changes to methods using line information. For each method we gathered two more pieces of information: the count of lines changed in the cloned portions and the count of lines changed in the non-cloned portions.

Storage of Methods: At this stage, we have obtained all the necessary pieces of information for all methods belonging to a particular revision. We store the methods in an XML file with an individual entry for each method. For each revision we create a separate XML file containing the methods of the corresponding revision. A file name is constructed by appending the revision number at its end so that we can generate it when necessary (for getting previously stored methods of the unchanged files of a former revision and for calculating dispersion).

Method Mapping: For mapping methods between two consecutive revisions we followed the technique proposed by Lozano and Wermelinger [18] (described in Section 2). We store the mapping information for each two consecutive revisions in a separate file. Method mapping was accomplished using method ids. The file names contain the revision numbers in a disciplined way so that we can generate them when necessary.

Calculation of Change Dispersion: To calculate change dispersion we examine each method genealogy and during this examination we update

four counters: **(i)** the count of method genealogies containing cloned code (C) **(ii)** the count of method genealogies containing non-cloned code (N) **(iii)** the count of method genealogies that have some changes in their cloned portions (C_c), and **(iv)** the count of method genealogies that have some changes in their non-cloned portions (N_c).

For each method genealogy, we first determine the revision in which the method was created. Starting from this revision we examine each method instance in subsequent revisions while the method remains alive. While processing a method genealogy we update the counters in the following way.

- If one or more of the method instances contains cloned code, we increment C by 1.
- If one or more of the method instances contains non-cloned code, we increment N by 1.
- If one or more of the method instances were changed in their cloned portions, we increment C_c by 1.
- If one or more of the method instances were changed in their non-cloned portions, we increment N_c by 1.

When processing a method genealogy it is the case that each of the counters can be at most incremented by 1.

5. Experimental Setup

Setup for CCFinderX: We set CCFinderX to detect clone blocks with a minimum size of 30 tokens with a TKS (minimum number of distinct types of tokens) set to 12 (as the default).

Setup for NiCad: Using NiCad [3] we detected block clones with a minimum size of 5 LOC in the pretty-printed format that removes comments and formatting differences. We used the NiCad settings in Table 3 for detecting three types of clones. The dissimilarity threshold means that the clone fragments in a particular clone class may have dissimilarities up to that particular threshold value between the pretty-printed and/or normalized code fragments. For all the settings in Table 3, NiCad was shown to have high precision and recall [23]. These settings for NiCad are considered standard [25, 26, 27, 28] for detecting the three types of clones. Before using

Table 2: SUBJECT SYSTEMS

	System	Domain	LER	SRev	ERev
Java	DNSJava	DNS protocol	23,373	2	1635
	Ant-Contrib	Web Server	12,621	5	176
	Carol	Game	25,092	2	1699
	jabref	Project Management	79,853	2	32
C	Ctags	Code Def. Generator	33,270	2	774
	Camellia	Multimedia	85,015	1	55
	QMail Admin	Mail Management	4,054	5	317
	Gnumakeuniproc	Project Building	83,269	689	799
C#	GreenShot	Multimedia	37,628	4	999
	ImgSeqScan	Multimedia	12,393	2	73
	Capital Resource	Database Management	75,434	1	122
	MonoOSC	Formats and Protocols	18,991	2	355
Python	Noora	Development Tool	14,862	4	140
	Marimorepy	Collection of Libraries	13,802	1	49
	Pyevolve	Artificial Intelligence	8.809	3	200
	Ocemp	Game	57,098	5	438

SRev = Starting Revision **ERev** = End Revision
LER = LOC in End Revision

Table 3: NICAD SETTINGS

Clone Type	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

the NiCad output for the Type-2 and Type-3 cases, we processed them in the following way.

1. Every Type-2 clone class that exactly matches a Type-1 clone class was excluded from the Type-2 outputs.

2. Every Type-3 clone class that exactly matches a Type-1 or Type-2 clone class was excluded from the Type-3 outputs.

Subject Systems: Table 2 lists the subject systems that were included in our study. Most of the systems except Carol[5] and Capital Resource[6] were downloaded from Sourceforge[7]. We selected this set of subject systems because they are diverse, differ in size, span 13 different application domains, and cover three different programming languages.

6. Experimental Results and Discussion

In this section, we present our experimental results and answer our research questions (presented in Table 1) on the basis of these results.

6.1. ***RQ 1: Is the change dispersion in cloned code higher than the change dispersion in non-cloned code?***

Motivation. Our central goal of this research work is to determine whether cloned code has any negative effect(s) on software evolution. For this purpose, we perform a comparative study of change dispersion in cloned and non-cloned code. If change dispersion in cloned code appears to be higher compared to that of non-cloned code, then it is likely that cloned code negatively affects software evolution and maintenance.

Methodology. To answer research question RQ 1 and the next two research questions (RQ 2 and RQ 3), we first determine the change dispersion of cloned (CD_c) and non-cloned code (CD_n) for each subject system using Eq. 1 and Eq. 2, respectively. The magnitude of CD_c and CD_n for each of the systems with associated decision points is shown in Table 4 (NiCad results) and in Table 5 (CCFinderX results). CCFinderX cannot detect clones in Python systems; however, including this clone detector in our study strengthens our language centric decisions, as seen in the next section when

[5]Carol: http://websvn.ow2.org/listing.php?repname=carol

[6]Capital Resource: http://www.ohloh.net/p/capitalresource/enlistments

[7]SourceForge: http://sourceforge.net

Table 4: DISPERSION OF CHANGES USING NICAD RESULTS

		Type 1 (NiCad)			**Type 2 (NiCad)**			**Type 3 (NiCad)**		
	System	CD_c	CD_n	DP	CD_c	CD_n	DP	CD_c	CD_n	DP
Java	DNSJava	24.53	5.91	$\ominus$	15.17	7.82	$\ominus$	18.16	7.55	$\ominus$
	Ant-Contrib	17.64	1.63	$\ominus$	2.22	1.95	$\ominus$	5	1.97	$\ominus$
	Carol	5.87	19.50	$\oplus$	9.35	19.33	$\oplus$	18.92	19.72	$\otimes$
	jabref	11.23	20.43	$\oplus$	8.95	21.78	$\oplus$	13.05	18.44	$\oplus$
C	Ctags	0	10.04	$\oplus$	20	9.66	$\ominus$	14.53	9.78	$\ominus$
	Camellia	0	9.85	$\oplus$	12.5	9.55	$\ominus$	35	8.76	$\ominus$
	QMail Admin	50	7.29	$\ominus$	42.85	8.03	$\ominus$	60	8.15	$\ominus$
	Gnumakeuniproc	12.5	0.42	$\ominus$	0	0.50	$\oplus$	1.38	0.51	$\ominus$
C#	GreenShot	8.88	29.32	$\oplus$	22.96	29.49	$\oplus$	30.37	30.47	$\otimes$
	ImgSeqScan	0.0	3.76	$\oplus$	0.0	3.73	$\oplus$	0.0	3.72	$\oplus$
	Capital Resource	0.0	4.92	$\oplus$	0.0	4.79	$\oplus$	3.95	4.47	$\oplus$
	MonoOSC	3.17	10.48	$\oplus$	5.26	10.42	$\oplus$	39.13	10.03	$\ominus$
Python	Noora	20.12	17.88	$\ominus$	14.52	21.81	$\oplus$	23.72	17.44	$\ominus$
	Marimorepy	2.32	15.83	$\oplus$	0	16.66	$\oplus$	37.5	15.12	$\ominus$
	Pyevolve	25.49	25.11	$\otimes$	25.0	30.12	$\oplus$	22.98	27.39	$\oplus$
	Ocemp	22.94	66.59	$\oplus$	42.28	65.54	$\oplus$	38.67	62.45	$\oplus$

CD_c= Dispersion of Changes in Cloned Methods

CD_n= Dispersion of Changes in Non-cloned Methods

DP = Decision Point: $\oplus = CD_n > CD_c$ $\ominus = CD_n < CD_c$ $\otimes = CD_n \approx CD_c$

addressing research question RQ 3. The tables contain 60 *decision points* in total.

Decision Point: Based on the change dispersion of non-cloned code (CD_n) and cloned code (CD_c) we make a decision as to whether the changes are more dispersed in the cloned code, more dispersed in the non-cloned code, or similarly dispersed. These decision points are determined for each system and for each of the following four clone cases: Type 1 (NiCad), Type 2 (NiCad), Type 3 (NiCad), and CCFinder. The decision points are shown in the dispersion tables under the column *DP* and are one of the following:

Category 1 (indicated with a $\oplus$). Changes to the non-cloned code are more dispersed than changes to the cloned code (i.e., $CD_n > CD_c$).

Category 2 (indicated with a $\ominus$) Changes to the non-cloned code are less dispersed than changes to the cloned code (i.e., $CD_n < CD_c$).

Category 3 (indicated with a $\otimes$) There is no substantive difference between the change dispersion of non-cloned and cloned code; the changes are similarly dispersed in both the non-cloned and the cloned code (i.e., $CD_n \approx CD_c$).

To determine whether the difference between the dispersions of cloned and non-cloned code of a subject system for a particular case is significant we calculated an *EligibilityValue* on the observed dispersions according to the following equation.

$$EligibilityValue = \frac{(CD_n - CD_c) \times 100}{\min(CD_n, CD_c)} \tag{4}$$

$$DecisionPoint = \begin{cases} \oplus, & \text{if } EligibilityValue > threshold \\ \ominus, & \text{if } EligibilityValue < -threshold \\ \otimes, & \text{otherwise} \end{cases}$$

We use a threshold of 10 when considering if the difference between dispersions is significant. An eligibility value between −10 and 10 inclusive is considered insignificant (i.e., $-10 \leq EligibilityValue \geq 10$). When *EligibilityValue* is greater than the threshold value (i.e., $EligibilityValue > 10$), the subject system will be considered to belong to *Category 1*. When *EligibilityValue* is less than the negative of the threshold value (i.e., $EligibilityValue < -10$), the subject system will be considered to belong to *Category 2*.

Rationale behind Selection of Threshold. We selected the calculation procedure and the threshold value to force a subject system with

Table 5: DISPERSION OF CHANGES USING CCFINDERX RESULTS

	Systems	CD_c	CD_n	DP
Java	DNSJava	18	7	$\ominus$
	Ant-Contrib	5	0	$\ominus$
	Carol	23	7	$\ominus$
	jabref	31	8	$\ominus$
C	Ctags	21	10	$\ominus$
	Camellia	30	9	$\ominus$
	QMail Admin	50	12	$\ominus$
	Gnumakeuniproc	1.26	2.73	$\oplus$
C#	GreenShot	14	5	$\ominus$
	ImgSeqScan	25	0.55	$\ominus$
	Capital Resource	3.95	4.58	$\oplus$
	MonoOSC	39.44	12	$\ominus$

CD_c= Dispersion of Changes in Cloned Methods

CD_n= Dispersion of Changes in Non-cloned Methods

DP = Decision Point

$\oplus = CD_n > CD_c$ $\quad \ominus = CD_n < CD_c$ $\quad \otimes = CD_n \approx CD_c$

** CCFinderX cannot detect clones in Python systems.*

large but very near dispersions to be placed in *Category 3*. For example, if CD_n = 41 and CD_c = 40, then *EligibilityValue* = $((41 - 40) \times 100)/40 = 2.5$, which is between −10 and 10 and considered insignificant. A subject system with small but nearly identical dispersions will be placed in either *Category 1* or *Category 2*, which is to be expected. For example, if CD_n = 3 and CD_c = 4, then *EligibilityValue* = $((3 - 4) \times 100)/3 = -33.33$, which is less than −10 and *Category 2* (i.e., $\ominus$) will be chosen.

Among 60 decision points contained in the tables, Table 4 and Table 5, 57 points fall in *Category 1* or *Category 2*. We call them *significant decision points* because the differences between the dispersions for these points are significant according to the *Eligibility Value* of Eq. 4. We ignored the remaining 3 decision points marked with a ($\otimes$).

According to 49.12% of the significant points, dispersion of changes in cloned code is less than the dispersion of changes in non-cloned code. The opposite is true for the remaining 50.88% points. Although the difference between the percentages is not significant, it indicates that, *the changes in the cloned portions of our investigated subject systems are sometimes more scattered than the changes in the non-cloned portions.*

Answer to RQ 1: In answer to RQ 1 we can state that *change dispersion in cloned code can sometimes be higher than the change dispersion in non-cloned code.* Considering the equations (Eq. 1 and Eq. 2) for calculating change dispersion we can, more specifically, say that *the percentage of methods affected by the changes in cloned code is sometimes greater than the percentage of methods affected by the changes in the non-cloned code.* This scenario indicates that the percentage of the cloned regions affected (by changes) during the evolution of a subject system is sometimes greater than the percentage of the affected non-cloned regions. Thus, cloned code is (sometimes) expected to exhibit higher change-proneness compared to non-cloned code.

6.2. ***RQ 2: Do different types of clones exhibit different change dispersion? If so, which type(s) of clones shows higher change dispersion compared to the others?***

Motivation. From the answer to the first research question we realize that cloned code sometimes negatively affects software evolution by exhibiting higher change dispersion compared to non-cloned code. This phenomenon

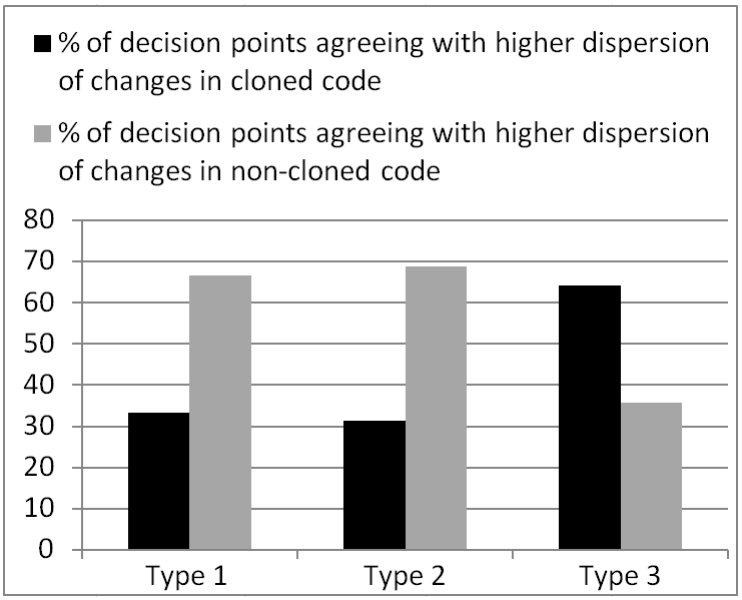

Figure 2: Overall type centric analysis of NiCad results

raised the question whether different types of clones are equally likely to exhibit higher change dispersion than non-cloned code. We find the answer to the question in the following way.

Methodology. We perform a clone-type centric analysis of the change dispersion results for answering this research question. Our type centric analysis depends only on NiCad results, since CCFinderX does not make a distinction between different types of clones. Considering all the significant decision points belonging to a particular clone type (Type 1, Type 2 and Type 3) in Table 4 we calculate two measures: **(1)** the proportion of decision points indicating that there is a higher dispersion of changes in cloned code, and **(2)** the proportion of decision points indicating that there is a lower dispersion of changes in cloned code. We plot these two measures for each clone type in Fig. 2. According to this graph, for most of the subject systems the dispersion of changes in Type 3 clones is higher than the dispersion of changes in the corresponding non-cloned code. However, the opposite is observed for the other two clone types. From these results we determine that Type 3 clones are likely more change-prone than Type 1 and Type 2 clones.

Using the data in Table 4 we plot another bar chart (Fig. 3) that shows

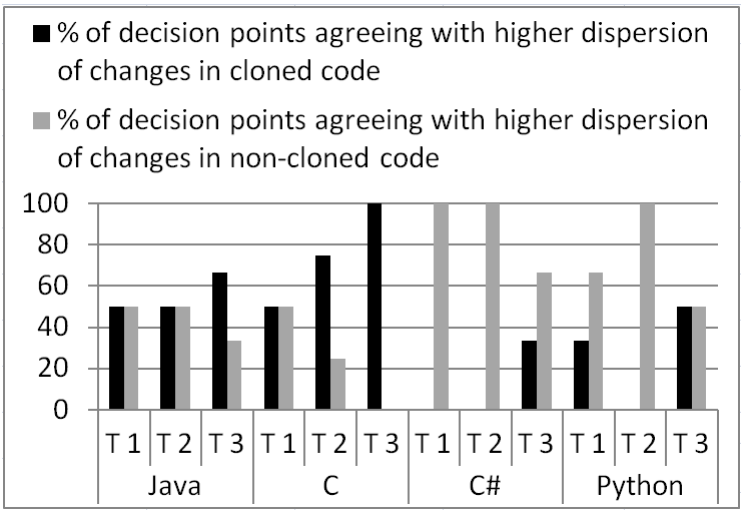

Figure 3: Type centric analysis of NiCad results by programming language

the type-wise change dispersion for each programming language. Considering the significant decision points belonging to a particular clone type and a particular language in Table 4 we measure: (1) the percentage of decision points indicating a higher dispersion of changes in cloned code, and (2) the percentage of decision points indicating a higher dispersion of changes in non-cloned code. According to this chart, in case of both Java and C systems, each type of clone has the probability of having more dispersed changes compared to the corresponding non-cloned code. We also see that Type 3 clones have the highest possibility of having more dispersed changes compared to the other two clone types for these two languages (Java and C). However, for the other two programming languages (C# and Python) cloned code has a lower likelihood of having more dispersed changes in general, except for Type 3 clones in Python.

Answer to RQ 2: In answer to *RQ 2* we can state that *Type 3 clones generally exhibit a higher likelihood of having more dispersed changes (than non-cloned code) compared to the other two clone types (Type 1 and Type 2).* So, it is expected that *Type 3 clones likely exhibit higher change-proneness and also require more maintenance effort compared to the other two types of clones.*

We also investigated the reason why the changes in Type 3 clones are

more dispersed compared to the other two clone types. According to our analysis, a possible reason is that the average number of clone fragments in a Type 3 clone class is generally higher than the average number of clone fragments per clone class of each of the other two clone types. A higher number of clone fragments in a clone class increases the possibility of dispersed or scattered changes, because clone fragments in a particular class often require corresponding changes and thus have a tendency of changing together. For further clarification we present the following example.

Suppose there are two clone classes, *Class1* and *Class2*. *Class1* has five clone fragments and *Class2* contains ten clone fragments. If a change occurs in a particular clone fragment in a particular class (*Class1* or *Class2*), the change might need to be propagated to the other clone fragments in the same class for ensuring consistency among the fragments. The clone fragments in a clone class might belong to different methods. In this situation, while a consistency ensuring change in *Class1* can affect at most five different methods, such a change in *Class2* can affect ten different methods and, in this way, *Class2* would exhibit a higher change dispersion.It is important to note that we measure change dispersion of a code region by determining the percentage of methods affected by changes in that region during evolution.

Considering the above example we infer that as Type 3 clone classes have a higher number of clone fragments (on average) compared to the clone classes of the other two clone types, Type 3 clones will exhibit a higher change dispersion than the other two types. Fig. 10 and Fig. 11 together present an example of the consistency ensuring changes in a Type 3 clone class. The Type 3 class in the example contains four clone fragments, which are four different methods. These four methods were changed consistently by adding the same parameter to each of the methods in a commit on revision 36. Each of the figures (Fig. 10 and Fig. 11) shows changes to two methods. The figures and their detailed descriptions are presented in Section 6.6.

Table 6 shows the average number of clone fragments per clone class for each clone type in each of the subject systems. For determining the average for a particular clone type of a particular subject system, we considered all the clone classes of that particular clone type in all of the revisions. From the table we see that the average number of clone fragments in a Type 3 clone class (ANCF 3) is most often higher (ten systems out of sixteen) than the average number of clone fragments in the other types of clone classes (ANCF 1 and ANCF 2).

Table 6: AVERAGE NUMBER OF CLONE FRAGMENTS PER CLONE CLASS TYPE

	System	ANCF 1	ANCF 2	ANCF 3
Java	DNSjava	2.44	2.31	2.89
	Ant-Contrib	2.22	2.66	3.01
	Carol	2.40	2.14	3.30
	JabRef	2.23	2.6	2.99
C	Ctags	2.47	2.45	2.9
	Camellia	2.49	3.23	2.94
	QMailAdmin	3.20	3.77	5.9
	Gnumakeuniproc	2.20	2.87	3.0
C#	GreenShot	3.69	3.14	3.03
	ImgSeqScan	2.00	2.34	2.89
	CapitalResource	2.71	3.00	3.05
	MonoOSC	4.52	2.00	2.44
Python	Noora	2.61	4.86	4.01
	Marimorepy	3.18	3.13	3.17
	Pyevolve	2.21	3.90	3.76
	Ocemp	2.55	2.41	2.73

ANCF 1 = Average number of clone fragments per Type 1 clone class
ANCF 2 = Average number of clone fragments per Type 2 clone class
ANCF 3 = Average number of clone fragments per Type 3 clone class

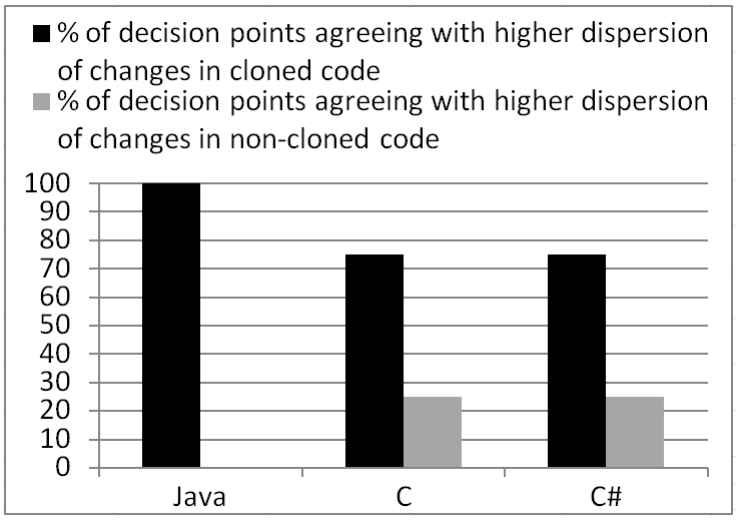

Figure 4: Programming language centric statistics of CCFinderX results

6.3. ***RQ 3: Do cloned and non-cloned code in the subject systems of different programming languages show different comparative scenarios of change dispersion?***

Motivation. Answering this question is important, because if the clones in a system written in a particular programming language generally show higher change dispersion than non-cloned code, then it is likely that clones in systems written in that language will be more change-prone, and it may be desirable to give clone management tasks a higher priority for these systems.

Methodology. In order to answer this research question, we perform a language centric analysis of our experimental results. Table 4 and Table 5, present the language centric statistics of the dispersion of changes occurring in the cloned and non-cloned code. Considering the significant decision points belonging to a particular programming language in a particular table we measured two proportions: **(1)** the proportion of significant decision points that indicate there is higher change dispersion in cloned code, and **(2)** the proportion of significant decision points that indicate there is higher change dispersion in non-cloned code.

According to the language centric statistics in Fig. 4 obtained from the CCFinderX results (Table 5), for Java, C and C#, most of the decision points

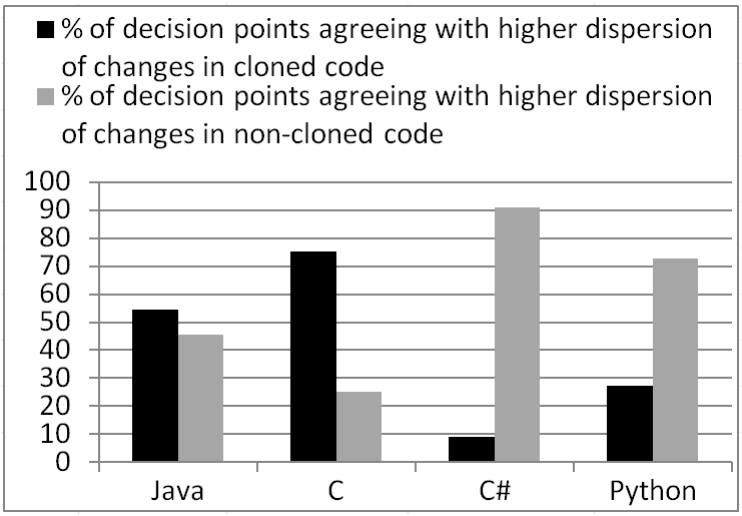

Figure 5: Programming language centric statistics of NiCad results

indicate that there is a higher dispersion of changes in cloned code than in non-cloned code.

The language statistics that we obtain from the NiCad results are shown in Fig. 5. According to this figure, a higher proportion of decision points belonging to both Java and C indicate that there is a higher dispersion of changes in cloned code compared to non-cloned code. However, the opposite is observed for the other two programming languages, C# and Python. For each of these two languages, a higher proportion of the significant decision points indicate there is a lower dispersion of changes in cloned code than in non-cloned code.

Answer to RQ 3: Considering Fig. 4 and Fig. 5 we can answer *RQ 3* by stating that *cloned code in subject systems written in Java and C often exhibit a higher change dispersion than non-cloned code. Thus, it is expected that cloned code in Java and C systems is more change-prone than non-cloned code. However, clones in C# and Python systems are not likely to show a higher change dispersion than non-cloned code. Thus cloned code in these systems (written in C# and Python) are expected to be less change-prone than non-cloned code.*

It is very difficult to explain why the comparative dispersion scenario of cloned and non-cloned code is different for different programming languages.

A possible reason could be the difference in the application domains of the subject systems. Several other factors such as: programmer's expertise and programmer's knowledge about the subject system can play important roles in controlling the comparative scenario between the change-proneness as well as change dispersion of cloned code and non-cloned code. For example, while making changes to a particular clone fragment if a maintenance programmer has the proper knowledge about the other similar clone fragments, they might decide to make changes to those clone fragments as well and eventually the dispersion of changes in clones will increase. On the other hand, a programmer that does not know that other clone fragments exist, may make changes to only a single clone fragment and in this way clones might gradually appear to exhibit less dispersed changes compared to non-cloned code. However, in this research we have not considered these factors while determining the change dispersion of cloned and non-cloned code.

6.4. ***RQ 4: Is higher dispersion of changes an indicator of higher instability (or change-proneness) in the source code?***

Motivation. We have already mentioned that higher change dispersion is a possible indicator of increased effort for understanding and analyzing the consequences (or impacts) of changes. In this subsection we investigate whether higher change dispersion indicates higher instability (i.e., change-proneness) of source code because, if higher change dispersion indicates higher code instability, then higher change dispersion should also be an indicator of higher maintenance effort.

Methodology. For the purpose of our investigation we determined the correlation between two measures, **(1)** Change Dispersion in the Entire Codebase ($CDEC$) and **(2)** Change-proneness (or instability) of the Entire Codebase (CEC). If these two measurements are correlated we can say that high $CDEC$ is a possible indicator for high CEC. We calculate $CDEC$ according to Eq. 3. The CEC of a subject system is calculated as follows.

Calculation of Change-proneness (or instability) of the Entire Code-base (*CEC*): For each of the subject systems, we determined the average number of changes that occurred per commit operation considering only those commits where there were some changes to the source code. The definition of change is presented in Terminology (Section 2).

After calculating the CEC and $CDEC$ for all candidate systems we determined whether the distribution of these two measures are normal. According to our investigation, the distribution of change dispersion ($CDEC$) is normal

Table 7: SPEARMAN CORRELATION BETWEEN CHANGE DISPERSION AND CODE INSTABILITY

	System	Dispersion	Instability
Java	DNSjava	33.16	6.88
	Ant-Contrib	2.9	2.41
	Carol	18.7	10.07
	JabRef	35.97	12.11
C	Ctags	41.85	4.95
	Camellia	67.1	25
	QMailAdmin	49.16	20.67
	Gnumakeuniproc	2.79	5.1
C#	GreenShot	28.52	4.39
	ImgSeqScan	3.7	5.27
	CapitalResource	4.72	4.92
	MonoOSC	27.15	4.77
Python	Noora	23.45	11.89
	Marimorepy	16.52	3.81
	Pyevolve	32.38	4.36
	Ocemp	67.62	20.68
Correlation coefficient between dispersion and instability = 0.60			

Table 8: Significance Test Details of Correlation Coefficient

Significance test details	
Correlation Coefficient =	0.60
Sample Size =	16 (As there are 16 subject systems)
t-Value =	2.81
Degrees of freedom =	14
Two-tailed probability value (p-value) =	0.013
If a probability value is less than 0.05, the corresponding correlation coefficient is considered significant.	
Thus, ***Correlation is significant at the 0.05 level (2-tailed). It reflects a true (rather than due to chance) correlation between change dispersion and code instability.***	

but, code instability (CEC) does not seem to be normally distributed. For this reason, we determined a non-parametric correlation, *Spearman Rank Correlation*, between these two measures. The details of correlation are shown in Table 7. We see that the correlation coefficient is positive (coefficient = 0.60) and it indicates the tendency that *high change dispersion ($CDEC$) is often (for about 60% of the cases) associated with high code instability (CEC)*. We have also calculated the significance[8] of this correlation coefficient. The details of the significance test are presented in Table 8. The *two-tailed* probability value regarding this test is 0.013 which is less than 0.05. Thus, the observed correlation coefficient is statistically significant and it reflects a true correlation[9] (rather than due to chance) between change dispersion and source code instability.

Answer to RQ 4: In answer to *RQ 4* we can say that *higher change dispersion often indicates higher instability (i.e., change-proneness) of source code. Also, as higher change-proneness indicates increased maintenance effort, higher change dispersion can possibly be regarded as an indicator of increased maintenance effort as well.*

6.5. RQ 5: If cloned code has higher change dispersion than non-cloned code, then does it also indicate that cloned code is more change-prone (i.e., unstable) than non-cloned code?

Motivation. From the answers to the first (*RQ 1*), second (*RQ 2*), and third (*RQ 3*) research questions it is clear that changes in cloned code are sometimes more dispersed compared to the changes in non-cloned code and, more specifically, clones in both Java and C systems have a higher probability of getting more dispersed changes compared to non-cloned code. From the answer to the fourth research question (*RQ 4*) we understand that higher change dispersion is a possible indicator of higher change-proneness. From this scenario we suspected that cloned code in the subject systems written in Java and C might have higher change-proneness (i.e., instability) compared to the non-cloned code in these systems. We investigated this matter considering method level granularity. For each of the candidate systems written in Java and C, we determined whether the presence of clones in methods causes these methods to become more change-prone compared to the methods that do not contain any clone fragments.

[8]Significance of correlation coefficient: `http://vassarstats.net/corr_rank.html`

[9]Implications of Correlation: http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf

Methodology. In this investigation we used the combined type clone results of NiCad. We can get clone detection results of NiCad in two ways: **(1)** obtaining the results of three types of clones (Type 1, Type 2, and Type 3) separately and **(2)** obtaining the results by combining these three types of clones. In the previous investigations we calculated dispersions for three types clones separately. However, in the following investigations we used the combined type results of NiCad to determine the combined effect of clones on code instability (i.e., change-proneness). CCFinderX provides us only with combined type results. While NiCad combines three types of clones (Type 1, Type 2, and Type 3) in the combined type result, CCFinderX provides only two types of clones (Type 1 and Type 2), since CCFinderX cannot detect Type 3 clones. We did not consider CCFinderX for the following investigations.

For the purpose of investigation, we at first separated the method genealogies detected in a candidate Java or C system into two disjoint sets: **(1)** cloned (fully cloned or partially cloned) method genealogies (CMG), and **(2)** fully non-cloned method genealogies (NMG). If any method instance in a particular method genealogy contains a clone, that genealogy is considered a member of CMG. On the other hand, if no method instance in a particular genealogy contains a clone, that genealogy is considered a member of NMG. Then, for the CMG and NMG sets we calculated an instability metric for methods.

The instability metric was calculated to quantify method change-proneness considering two things, **(1)** method longevity, and **(2)** method size. A method with high longevity has the probability of getting more changes than a method with comparatively low longevity. Also, method size might have an effect on method instability. We normalized the effects of method size and longevity in the instability metric in the following way.

Method Instability considering Longevity and Size (MILS): For a particular set of method genealogies, we calculate the average number of changes received per 100 LOC (Lines of Code) of a method instance per 100 commit operations. A particular method genealogy consists of several method instances. For both sets of genealogies (the CMG set and the NMG set) we calculate $MILS_{CMG}$ and $MILS_{NMG}$ in the following way.

$$MILS_{CMG} = \frac{NOC_{CMG} \times 10000}{|CMG| \times ALS_{CMG} \times AS_{CMG}} \tag{5}$$

NOC_{CMG} denotes the total number of changes that occurred to all method

instances of the method genealogies in the set CMG, ALS_{CMG} denotes the average life span per genealogy in CMG, and AS_{CMG} is the average size of a method instance in CMG. A particular method genealogy in CMG can have several method instances. While determining AS_{CMG} we at first find the summation of the sizes of all method instances of all genealogies. Then, we divide this sum by the total count of all method instances of all genealogies in CMG to get AS_{CMG}. We calculate ALS_{CMG} according to the following equation.

$$ALS_{CMG} = \frac{\sum_{g \epsilon CMG} LS(g)}{|CMG|} \tag{6}$$

Here, g is a cloned method genealogy in the set CMG. We know that the set CMG contains the cloned (fully or partially) method genealogies. $LS(g)$ is the life span of g. $LS(g)$ is the count of commit operations for which the genealogy g remained alive during evolution.

Justification of $MILS$ metric: Now we have a close look at Eq. 5. The term $NOC_{CMG}/(|CMG| \times ALS_{CMG})$ gives us the count of changes received by a method instance (of CMG) per commit operation. AS_{CMG} is the average size of a method instance. So, if we divide the term $NOC_{CMG}/|CMG| \times ALS_{CMG}$ by AS_{CMG}, we will get the count of changes that occurred per line of code of a method instance (of CMG) per commit operation. At last, by multiplying 10000 with the result we get the count of changes happened per hundred LOC of a method instance per hundred commit operations. Thus, Eq. 5 reasonably calculates $MILS_{CMG}$.

Similarly, we calculate $MILS_{NMG}$ according to the following equation.

$$MILS_{NMG} = \frac{NOC_{NMG} \times 10000}{|NMG| \times ALS_{NMG} \times AS_{NMG}} \tag{7}$$

A particular genealogy might not live for 100 commits and might not consist of 100 LOC. However, multiply by 10000 in the above equations just gives us larger values for $MILS_{CMG}$ and $MILS_{NMG}$ for better understanding. We perform the following three investigations using the $MILS$ metric.

(1) Investigation of the instability (i.e., change-proneness) of cloned and non-cloned methods: We calculated the values of $MILS_{CMG}$ and $MILS_{NMG}$ using the NiCad clone detector for each of the Java and C subject systems and plotted these values in Fig. 6. We see that for all systems but Camellia, $MILS_{CMG} > MILS_{NMG}$.

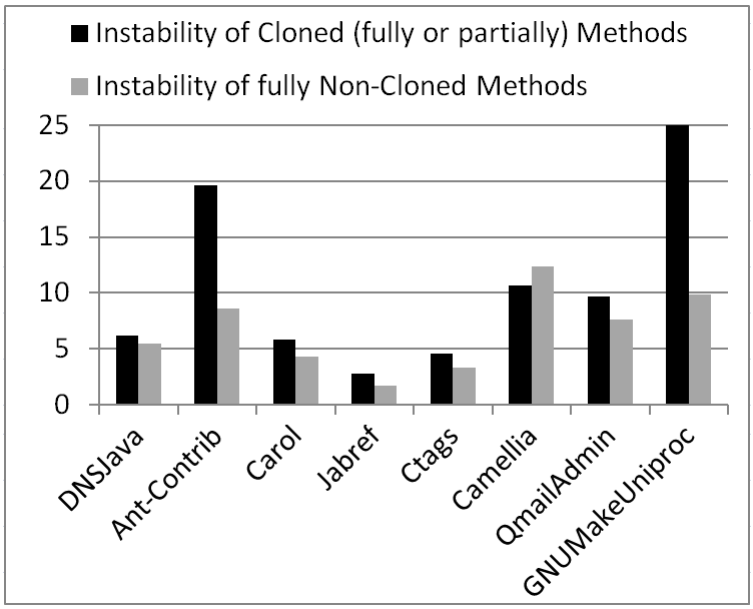

Figure 6: Instability of cloned (fully or partially) and non-cloned method genealogies using NiCad Results

Findings: Fig. 6 indicates that cloned methods in Java and C systems have more changes than non-cloned methods. In other words, the instability (i.e., change-proneness) of cloned methods for the subject systems written in Java or C is, most of the time, higher than the instability of non-cloned methods in these systems.

(2) Investigation of the instability (i.e., change-proneness) of fully cloned and partially cloned methods: In the previous investigation we observed that cloned methods (or cloned method genealogies (CMG)) exhibit higher instability compared to the non-cloned methods (or non-cloned method genealogies NMG)). We suspected clones to be a possible cause of this higher instability of cloned (fully or partially) methods. For determining whether clones are really responsible for higher instability of cloned methods, we considered the instability of fully cloned and partially cloned methods separately. Then, we made the following two comparisons.

- comparison of the instabilities of fully cloned and fully non-cloned method genealogies

- comparison of the instabilities of partially cloned and fully non-cloned method genealogies

The intuition behind these comparisons is that if both fully cloned and partially cloned methods exhibit higher instability compared to the fully non-cloned methods, clones can be a possible cause of the higher instability of cloned (fully or partially) methods.

The set CMG contains both fully cloned and partially cloned method genealogies. A method genealogy is considered to be a fully cloned genealogy if each method instance of this genealogy is fully cloned. On the other hand, in a partially cloned method genealogy there is at least one method instance that is not fully cloned. The sets of fully cloned and partially cloned genealogies are termed FCG and PCG respectively. For each of these sets we computed the value of the already defined instability metric $MILS$ according to the following equations.

$$MILS_{FCG} = \frac{NOC_{FCG} \times 10000}{|FCG| \times ALS_{FCG} \times AS_{FCG}} \tag{8}$$

$$MILS_{PCG} = \frac{NOC_{PCG} \times 10000}{|PCG| \times ALS_{PCG} \times AS_{PCG}} \tag{9}$$

We calculated $MILS_{FCG}$ and $MILS_{PCG}$ for each of the candidate systems using the NiCad clone detector. The comparison between the instabilities of fully cloned and fully non-cloned method genealogies has been shown in the graph of Fig. 7. According to this graph, *fully cloned methods exhibit higher instability compared to the fully non-cloned methods for all of the subject systems but Camellia.*

We compare the instabilities of fully non-cloned and partially cloned method genealogies in Fig. 8. We did not find any partially cloned methods in our subject system, GNUMakeUniproc. For this reason, this system does not have a black bar. According to this graph, *partially cloned methods appear to exhibit a higher probability of being more unstable (i.e., change-prone) than the fully non-cloned methods.*

Findings: From the above scenario we infer that clones are a possible cause of making the cloned (fully or partially) methods more unstable.

(3) Investigation of the commits having changes to the cloned portions of the cloned methods: We also investigated the commit operations of the Java and C subject systems to determine whether clones are

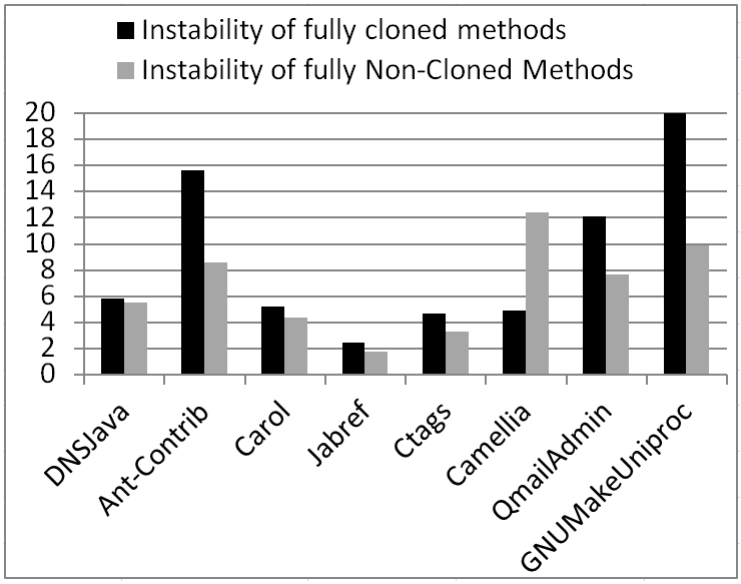

Figure 7: Instability of fully cloned and fully non-cloned method genealogies considering NiCad Results

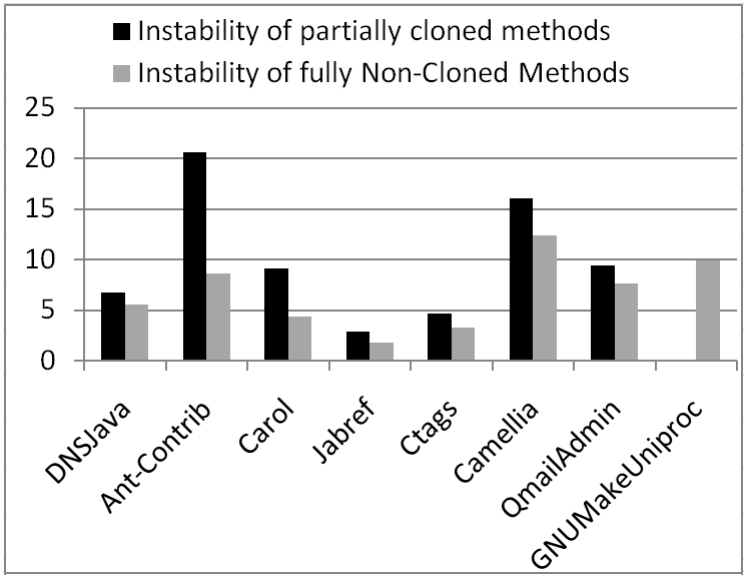

Figure 8: Instability of partially cloned and fully non-cloned method genealogies considering NiCad Results

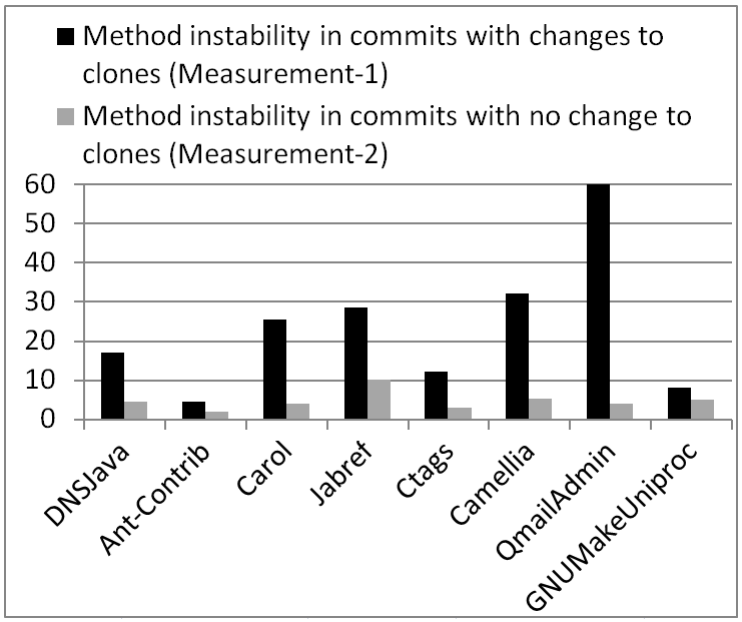

Figure 9: Average number of changes per commit operation considering NiCad Results

responsible for the higher instability of cloned methods. For the purpose of this investigation we separated the commit operations that occurred for a subject system into the following two disjoint sets.

Set 1: The commits with some changes to the cloned portions of the cloned methods are contained in this set.

Set 2: This set consists of the commits with no changes to the cloned portions of the cloned methods.

Then, we calculated the following two measures for these two sets.

Measurement-1: Average number of changes to methods (cloned or non-cloned) per commit operation of Set 1.

Measurement-2: Average number of changes to methods (cloned or non-cloned) per commit operation of Set 2.

We calculated the values of these two measurements using NiCad for each of the subject systems and show these values in Fig. 9. We see that for each of the subject systems, Measurement-1 is higher than Measurement-2. The subject system, QMailAdmin, shows the highest difference between these two measures. We performed the MWW (Mann Whitney Wilcoxon) test [33] on the observed values for these two measures to see whether there is a signif-

icant difference between Measurement-1 and Measurement-2. The p-value (probability value) of the test is 0.0018 which is less than 0.05 and it indicates that there is a significant difference between these two measures. Thus, Measurement-1 is significantly higher than Measurement-2. In other words, the average number of changes to the commits where there are some changes to the clones is significantly higher than the average number of changes to the commits with no changes to the clones.

Findings: From this we conclude that, changes to the cloned portions of methods are always associated with a higher amount of changes. Such a finding is consistent with that of Lozano and Wermelinger [18].

Answer to RQ 5: Considering the findings of the three investigations conducted for answering *RQ 5* we can state that *cloned code is more change-prone (or unstable) than non-cloned code. In other words, higher change dispersion in cloned code than in non-cloned code also indicates higher instability or change-proneness of cloned code compared to non-cloned code.*

6.6. ***RQ 6: How does change dispersion in clones affect the stability of cloned code?***

Motivation. From the answer to *RQ 5* we understand that higher change dispersion in cloned code also indicates higher instability of cloned code compared to non-cloned code. Based on this scenario we wanted to further investigate how higher change dispersion in clones increases the instability of cloned methods. We investigate in the following way.

Methodology. We manually investigated all of the changes occurred to the clones of our subject system Ctags. We choose this subject system because it is relatively small in size and has a reasonably long revision history.

Our investigation was based on the combined type clone results from NiCad. As indicated in Table 2, we analyze 774 revisions as well as commits of Ctags. We found 61 commit operations where there were some changes to the cloned portions of cloned methods. We separated these commits into two sets. One set consists of those commits each of which had changes to a single clone fragment only. We refer to this set as *Set-1*. The other set contains those commits each of which had changes to more than one clone fragments. We term this set *Set-2*. We found respectively 33 and 28 commits in Set-1 and Set-2. We consider that Set-2 contains dispersed changes to clones, because each of the commits in this set affects more than one clone fragment. We analyzed the commits in Set-2 to determine whether the clone fragments changed in a particular commit belong to the same clone class.

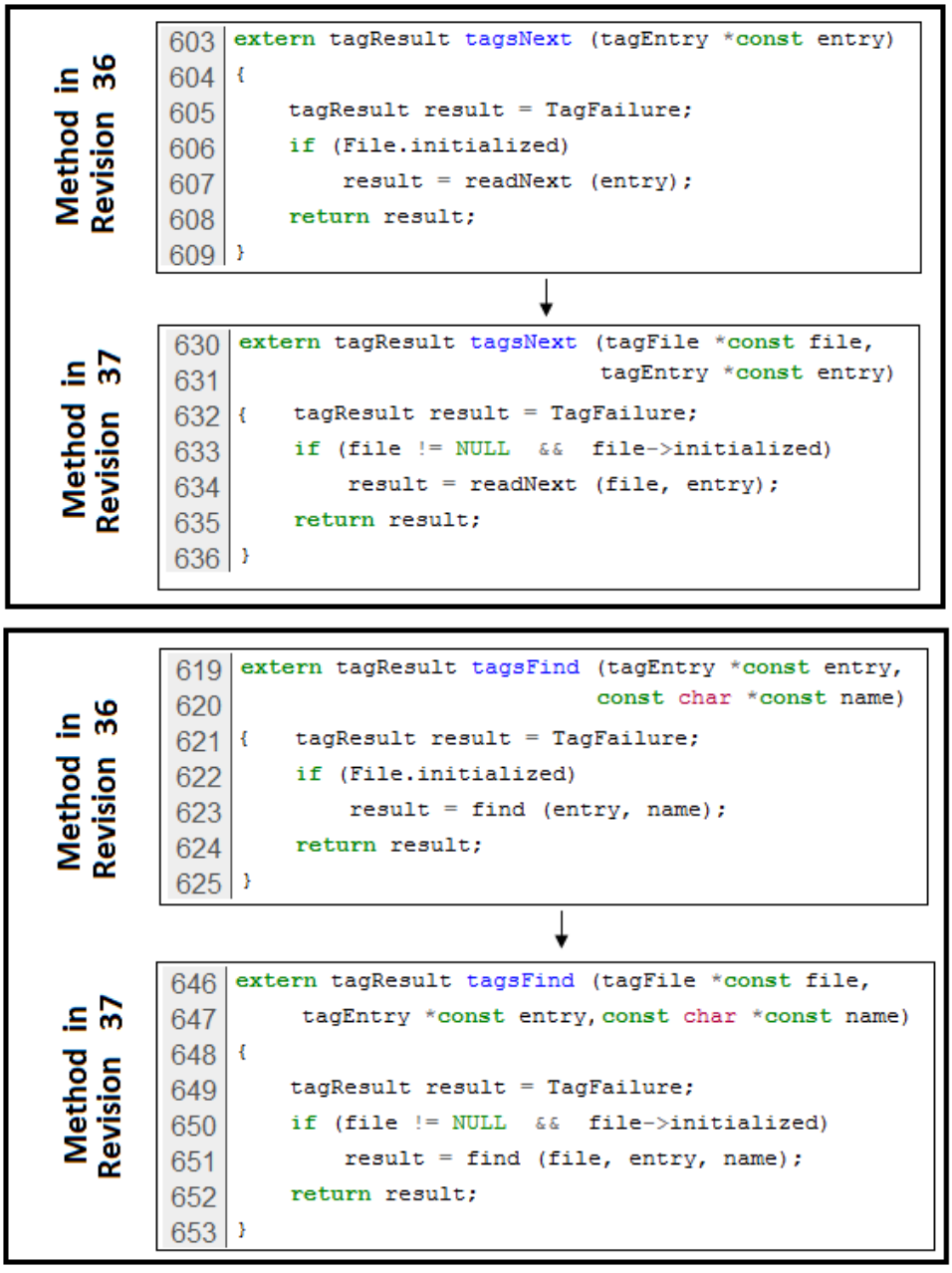

Figure 10: Changes in CTAGS from the commit operation applied to revision 36

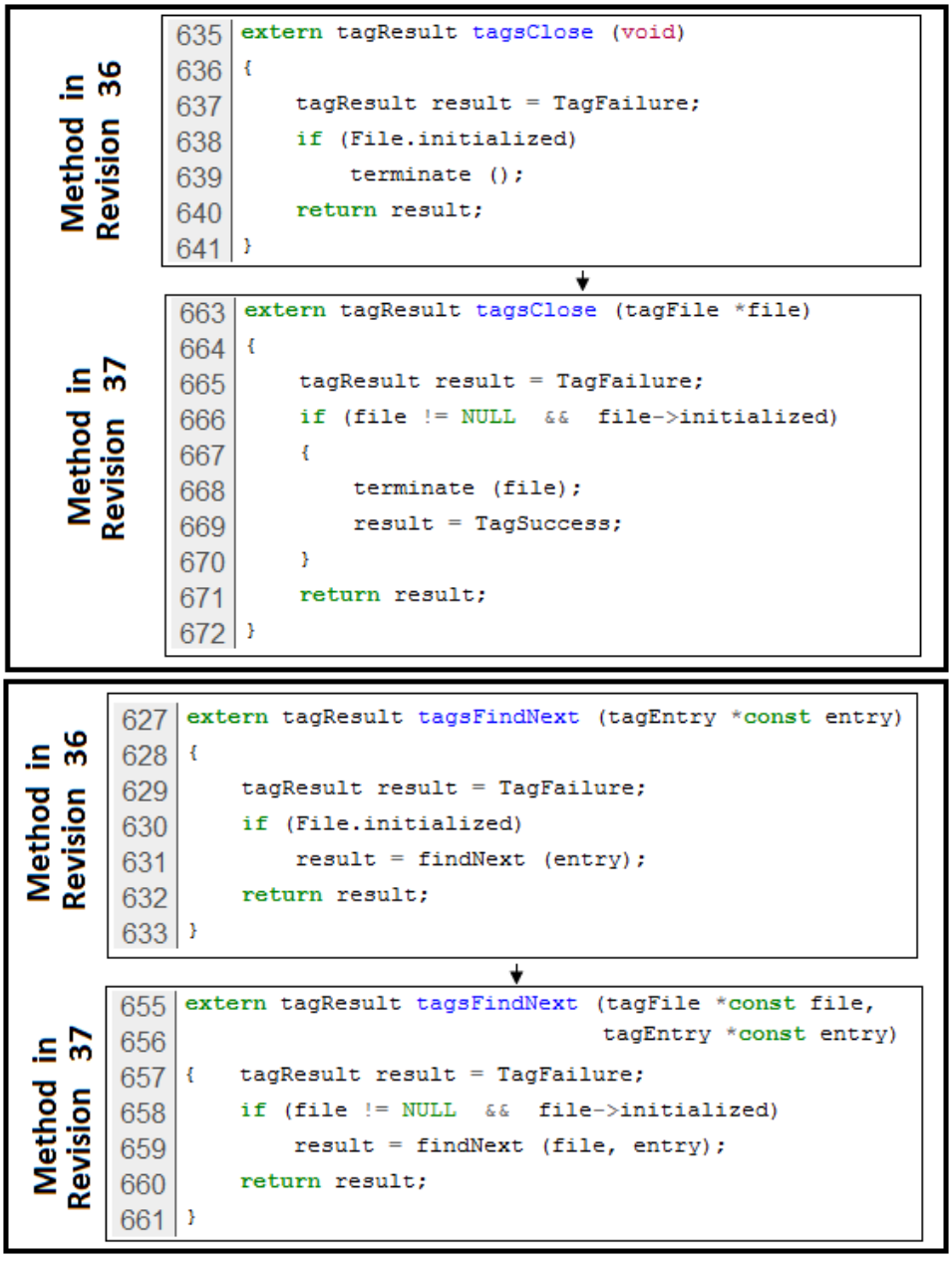

Figure 11: Changes in CTAGS from the commit operation applied to revision 36

Our intention was to find the reason why changes are happening to multiple clone fragments at a time and whether these changes are related.

We identified 19 commit operations in Set-2 where more than one clone fragment belonging to the same clone class were changed in the same way. The pattern of changes indicates that they were made to maintain consistency in the clone fragments. The changes in the commit operation applied to revision-36 are shown in Fig. 10 and Fig. 11. These figures contain four methods in revision 36 and their corresponding snap-shots in revision 37. Each figure shows the corresponding snapshots for two methods. According to NiCad result these four methods are four clone fragments (Type-3 clones) that belong to the same clone class in revision 36. The commit operation on revision 36 changed each of these methods by adding an extra parameter, *file*. We can easily understand the changes happened to the methods in revision 36 by comparing the corresponding snap-shots in revision 37. The changes imply that they were made to all the clone fragments for ensuring consistency. We have already defined (Section 2) such changes as *Consistency Ensuring Changes to Clones*. We consider these 19 commit operations (having Consistency Ensuring Changes to Clones) as a separate set and term this set CEC (the set of Consistency Ensuring Commits).

However, for each of the remaining 9 commit operations in Set-2, the clone fragments that received some changes belonged to different clone classes. In other words, no two clone fragments in such a commit belonged to the same clone class.

We observe that in all 61 commits (the commits with some changes to the cloned portions of the cloned methods) the cloned portions of the cloned methods received 175 changes in total. Among these changes, 85 (48.57%) changes occurred during the 19 commit operations in the set CEC. We see that CEC contains only 31.14% of the 61 commits. From this scenario we come to the conclusion that a major portion of the changes to the cloned code are made to ensure the consistency of the clone fragments. These consistency ensuring changes are responsible for higher change dispersion as well as higher change-proneness in cloned code and possibly require additional effort during maintenance.

We investigated the clone fragments appeared in the set CEC to see whether we can categorize the clone fragments that require consistency ensuring changes. We found that for 12 commits (63.15%) in CEC the clone fragments are full methods (not just a portion of method). For the remaining commit operations, the clone fragments were either condition blocks (if /

else) or case statements. From this we determine that consistency ensuring changes mainly occur to the fully cloned methods.

We further analyzed the clone fragments changed in commits of the set CEC to determine which type(s) of clones mainly require the consistency ensuring changes. According to our observation,

- 12 commits (63.15%) in CEC contained changes to the Type 3 clone fragments.
- 8 commits (42.1%) in CEC contained changes to the Type 2 clone fragments.
- 5 commits (26.31) in CEC contained changes to the Type 1 clone fragments.

From this scenario we see that the highest proportion of the consistency ensuring commits involve changes to the Type 3 clone fragments. However, this percentage for the Type 2 clones should also be taken into account. The lowest proportion of commits (in CEC) contained changes to the Type 1 clones. Possibly because of such a scenario in other systems, Type 3 clones appear to exhibit the highest probability of getting more dispersed changes compared to the other two types of clones (Fig. 2).

Answer to RQ 6: From our findings described above we answer *RQ 6* by stating that *consistency ensuring changes to clones are mainly responsible for higher change dispersion as well as higher change-proneness in cloned code. Because of these consistency ensuring changes, cloned code is expected to require additional maintenance effort.*

6.7. Discussion

Summary of Our Findings: From our answers to six research questions we understand that high change dispersion is an indicator of high change-proneness. Higher change dispersion in the code-base can also be an indicator of higher coupling among program entities. If two or more entities are coupled, that means related, changes in one entity might require corresponding changes in the other related entities[1]. In presence of higher coupling among program entities, changes will affect a higher number of entities and change dispersion will eventually be higher.

Cloned code, especially in the subject systems written in Java and C, often exhibits higher change dispersion than non-cloned code. Type 3 clones

exhibit the highest probability of having more dispersed changes (than non-cloned code) among three clone types (Type 1, Type 2, and Type 3). We also observed that cloned methods (both fully cloned and partially cloned) have higher instability (i.e., change-proneness) compared to non-cloned methods. We manually investigated whether and how higher change dispersion in cloned code is related to higher change-proneness. According to our investigation, *consistency ensuring changes to clones* (defined in Section 2) are primarily responsible for higher change dispersion as well as higher change-proneness in cloned code. We also observed (from one of our subject systems) that consistency ensuring changes mainly occur to the Type 3 clone classes. However, Type 1 and Type 2 clone classes also require such consistency ensuring changes.

Implications: Our findings imply that clone refactoring might help us in reducing change dispersion as well as change-proneness (or instability) in cloned code. More specifically, reduction of the number of clone fragments in a clone class will help in minimizing change dispersion in cloned code, because intuitively, a higher number of clone fragments in clone classes increases the possibility of consistency ensuring changes to the clone classes. In the case of refactoring we should primarily focus on Type 3 clone classes, because according to our observation, Type 3 clones required the highest proportion of consistency ensuring changes. Also, according to our findings, it might be a good idea to primarily consider the subject systems written in Java and C while making clone refactoring decisions. Type 1 and Type 2 clones should also not be ignored during refactoring. In general, refactoring of Type 1 clone classes might seem easier, because all the clone fragments in a Type 1 clone class are the same.

Effect of Cross-Cutting Concerns on Change Dispersion: Bruntink et al. [2] performed an in-depth investigation on the presence of clones in cross-cutting concerns. According to their observation, cross-cutting concerns are sometimes implemented using similar code fragments (i.e., clone fragments) and these clone fragments are scattered throughout the code-base. They reported that about 25% of the lines of code in a code-base can be dedicated to cross-cutting concerns. Because of corresponding changes to these cross-cutting concern clones, changes in cloned code might be more dispersed (or scattered) compared to non-cloned code. In the previous paragraph, we mentioned that clone refactoring is a way of minimizing change dispersion in cloned code. However, sometimes it might not be possible to decompose those clone fragments that implement cross cutting concerns from

the rest of the system [4]. In other words, refactoring of cross-cutting concern clones might not be possible. Thus, we feel the necessity of an automated system that will be able to keep track of all the clone classes and respective clone fragments (whether these clone fragments implement cross cutting concerns or not) so that while changing a particular clone fragment in a particular clone class the responsible programmer should also be notified about the other clone fragments in the same class, because these fragments might require corresponding changes.

Finally, we conclude by saying that our proposed measurement, change dispersion, can help us in fine grained analysis of clone impact and also can help us in deciding how to minimize negative impacts.

7. Threats to Validity

7.1. Threats to Construct Validity

We have measured the change dispersion of a particular code region (cloned region, non-cloned region, or the entire code base) by determining how many methods in that region have changed during evolution. With such a consideration, the changes occurred outside any method boundary have been disregarded in our experiment. Changes might also be dispersed on some other program entities such as structures, unions, and non-source-code files which have not been considered in our measurement.

However, we limited our experiment on the source code files only because, we wanted to compare the change-proneness of cloned and non-cloned code. Also, from our answer to the fourth research question (*RQ 4*) we observe that our measured change dispersion (i.e., the percentage of affected methods) is positively correlated with source code change-proneness of the entire code base and the correlation coefficient is statistically significant. From such an observation we believe that consideration of methods (only) for measuring change dispersion is reasonable.

7.2. Threats to External Validity

We calculated and analyzed the dispersion of changes of cloned and non-cloned code for only 16 subject systems, which is not a sufficient number of systems to make a general conclusion regarding different types of clones and programming languages. Also, some other important factors such as programmer expertise, application domain, and programmer knowledge about application domain were not considered in our experiment. However, our

selection of subject systems considering different application domains (thirteen domains), and programming languages (four programming languages) has considerably minimized these drawbacks, and thus we believe that our findings are significant.

The subject systems used in this study are relatively small to medium sizes. However, the systems have long version histories. Also, for the smaller systems (such as Ctags) we could perform an in-depth manual investigation regarding the effect of clones on method instability and change dispersion. Moreover, the systems are diverse in size, and thus, we believe that our experimental results and findings are not influenced by system size.

7.3. Threats to Internal Validity

CCFinderX cannot detect clones in Python code. Also, the two clone detectors do not provide entirely comparable results: NiCad provides clone results for three types of clones separately and CCFinderX provides only combined type results. However, the inclusion of CCFinderX strengthens our language centric decision results for both Java and C systems, since the CCFinderX decision results agree with NiCad's for these two languages.

For different clone detector settings the experimental results more specifically the type centric analysis outcome (i.e., regarding NiCad) can be different. However, the settings that we have used for NiCad are considered standard [25, 26, 27, 28]. Also, we use the default settings for CCFinderX. Thus, we believe that our experimental results and findings are reliable.

Our proposed metric, *change dispersion*, may not be perfectly representative of the maintenance efforts required for a particular code region. We have not measured source code change effort in this experiment. However, *change dispersion* is positively correlated with source code change-proneness and this correlation is statistically significant. From this we expect that *change dispersion* is a possible indicator of maintenance effort.

8. Related Work

Over the last several years, the impact of clones has been an area of focus for software engineering research resulting in a significant number of studies and empirical evidence. Kim et al. [12] proposed a model of clone genealogy. Their study with the revisions of two medium sized Java systems showed that refactoring clones may not always improve software quality. They also argued that aggressive and immediate refactoring of short-lived clones is not

required and that such clones might not be harmful. Saha et al. [29] extended their work by extracting and evaluating code clone genealogies at the release level of 17 open source systems involving four different languages. Their study reports similar findings to Kim et al. and concludes that most of the clones do not require any refactoring effort.

Kapser and Godfrey [11] strongly argued against the conventional belief that clones are harmful. In their study they identified different patterns of cloning and showed that about 71% of the cloned code has a kind of positive impact in software maintenance. They concluded that cloning can be an effective way of reusing stable and mature features.

Lozano and Wermelinger [18] developed a prototype tool to track the frequency of changes to cloned and non-cloned code with method level granularity. On the basis of their study on four open source systems they concluded that the existence of cloned code within a method significantly increases the required effort to change the method. In a recent study [17] they further analyzed clone imprints over time and observed that cloned methods remain cloned most of their life time and cloning introduces a higher density of modifications in the maintenance phase.

Juergens et al. [10] studied the impact of clones on large scale commercial systems and suggested that inconsistent changes occurs frequently with cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault. Aversano et al. [1] on the other hand, carried out an empirical study that combines clone detection and co-change analysis to investigate how clones are maintained during evolution or bug fixing. Their case study on two subject systems confirmed that most of the clones are consistently maintained. Thummalapenta et al. [32] in another empirical study on four subject systems concluded that most of the clones are changed consistently and other inconsistently changed fragments evolve independently.

In a recent study [5] Göde and Harder replicated and extended Krinke's study [14] using an incremental clone detection technique to validate the outcome of Krinke's study. They supported Krinke by assessing cloned code to be more stable than non-cloned code in general while this scenario reverses with respect to deletions.

Hotta et al. [9] studied the impact of clones by measuring the modification frequencies of cloned and non-cloned code of several subject systems. Their study using different clone detection tools suggests that the presence of clones does not introduce extra difficulties to the maintenance phase.

Krinke [13] measured how consistently the code clones are changed during

maintenance using *Simian* [31] and *diff* on Java, C and C++ code bases considering Type-I clones only. He found that clone groups changed consistently through half of their lifetime. In another experiment he showed that cloned code is more stable than non-cloned code [14]. In his most recent study [15] he calculated the average last change dates of the cloned and non-cloned code and observed that cloned code is more stable than non-cloned code.

Bruntink et al. [2] performed an in-depth investigation on the presence of clones in cross-cutting concerns. They showed that clone detection tools can also be used for detecting cross-cutting concerns, because some cross-cutting concerns are implemented using similar pieces of code.

None of the existing studies measured change dispersion. It is important to measure change dispersion because, it can help us in fine-grained analysis of the impact of a particular code region on maintenance. Our experimental results suggest that the changes in the cloned regions can sometimes be more dispersed than the changes in the non-cloned regions of a subject system. We have also investigated and reported the implications and causes of higher change dispersion in cloned code. This information can help us minimize the negative impact of clones on maintenance.

9. Conclusion

In this paper, we introduced a new metric *change dispersion* that measures the extent to which the changes in a particular code region (cloned, non-cloned, or the entire code-base) are scattered. The intuition is that higher dispersion of changes results in more maintenance effort and cost. We performed a fine grained empirical study using change dispersion with the primary goal of determining whether cloned code negatively effects software maintenance, and, if so, how can we minimize any negative effects. With this focus we answered six important research questions presented in Table 1. To answer these research questions we investigated 16 subject systems written in four different programming languages (Java, C, C# and Python) involving two clone detection tools (CCFinderX and NiCad) and considering three major types of clones (Type 1, Type 2, and Type 3). The answers to the research questions are presented in Table 9 and are summarized below.

Cloned code, especially in Java and C systems, often exhibits higher change dispersion compared to non-cloned code. Higher change dispersion is an indicator of higher change-proneness in the source code (change dispersion is positively correlated with source code change-proneness, Spearman's

Table 9: Answers to the Research Questions

Serial	Answers to the Questions (AQ)
Answers to the Questions Regarding Change Dispersion in Cloned and Non-cloned Code	
AQ 1.	Dispersion of changes in the cloned code is sometimes higher than the dispersion of changes in the non-cloned code. In other words, the percentage of methods affected by changes in cloned code is sometimes higher than the percentage of methods affected by the changes in non-cloned code. Thus, cloned code is possibly more change-prone than non-cloned code during the maintenance phase.
AQ 2.	Type-3 clones exhibit higher change dispersion compared to the Type-1 and Type-2 clones.
AQ 3.	Cloned code in the subject systems written in Java and C has a higher probability of getting more dispersed changes than non-cloned code. Thus the clones in Java and C systems are expected to require more maintenance effort than non-cloned code.
Answers to the Questions Regarding Change Dispersion and its Relation with Source Code Change-proneness (or Instability)	
AQ 4.	High change dispersion in source code is a possible indicator of high change-proneness (i.e., instability) in source code.
AQ 5.	In the case of subject systems written in Java and C we found that cloned code has higher change dispersion compared to non-cloned code. For each of these subject systems we also found cloned code to be more change-prone (i.e., unstable) than non-cloned code considering method level granularity. We observed that both fully cloned and partially cloned methods of Java and C systems exhibit higher instability compared to the fully non-cloned methods.
AQ 6.	Consistency ensuring changes to clones are mainly responsible for higher change dispersion as well as higher change-proneness in cloned code. According to our manual investigation of all the changes to the clones of CTAGS during its evolution, • A considerable amount (48.57%) of changes in the cloned code were made to ensure consistency of the clone fragments belonging to the same clone class. • Consistency ensuring changes mainly took place to the fully cloned methods. • A major proportion (63.15%) of the consistency ensuring changes occurred to Type 3 clone fragments.

Rank correlation coefficient = 0.60, and the correlation is statistically significant). Cloned code often shows higher change-proneness than non-cloned code. From this scenario we suspect that cloned code sometimes requires more effort and cost to be maintained than non-cloned code. We also observed that Type 3 clones exhibit the highest change dispersion among the three types of clones (Type 1, Type 2, and Type 3).

We also investigated the reason(s) behind higher change dispersion in cloned code. According to our manual analysis of all the changes that occurred to the clones in our subject system, CTAGS, we came to the conclusion that *consistency ensuring changes to clones* (defined in Section 2) are mainly responsible for higher change dispersion as well as higher change-proneness in cloned code.

Clone refactoring can be helpful in reducing change dispersion in cloned code however, clones can sometimes be a result of implementing cross-cutting concerns [2] and it might sometimes not be possible to refactor such clones [4]. From this we feel it is necessary to have an automated clone management system to track all the clone classes with their respective clone fragments so that when a programmer attempts to change a particular clone fragment they will be automatically notified about the presence of other clone fragments in the same class.

As future work we plan to investigate change dispersion in cloned and non-cloned code for different application domains to determine if different types of applications generally exhibit higher change dispersion.

References

[1] L. Aversano, L. Cerulo, M. D. Penta, "How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81 – 90.

[2] M. Bruntink, A. v. Deursen, R. v. Engelen, T. Tourwe, "On the Use of Clone Detection for Identifying Crosscutting Concern Code", *TSE*, 2005, pp. 804 – 818.

[3] J. R. Cordy, and C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC (Tool Demo Track)*, 2011, pp. 219 – 220.

[4] Cross Cutting Concerns: `http://en.wikipedia.org/wiki/Cross-cutting_concern`.

[5] N. Göde, and J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65 – 74.

[6] N. Göde, R. Koschke (2011), "Frequency and risks of changes to clones", Proc. *ICSE*, 2011, pp. 311 – 320.

[7] J. Harder, and N. Göde, "Cloned code: stable code", *Journal of Software: Evolution and Process*, 2013, 25(10): 1063 – 1088.

[8] A. E. Hassan. "Predicting Faults Using the Complexity of Code Changes", Proc. *ICSE*, 2009, pp. 78 – 88.

[9] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software", Proc. *EVOL/IWPSE*, 2010, pp. 73 – 82.

[10] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?", Proc. *ICSE*, 2009, pp. 485 – 495.

[11] C. Kapser, and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", *ESE*, 2008, 13(6): 645 – 692.

[12] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies", Proc. *ESEC-FSE*, 2005, pp. 187 – 196.

[13] J. Krinke, "A study of consistent and inconsistent changes to code clones", Proc. *WCRE*, 2007, pp. 170 – 178.

[14] J. Krinke, "Is cloned code more stable than non-cloned code?", Proc. *SCAM*, 2008, pp. 57 – 66.

[15] J. Krinke, "Is Cloned Code older than Non-Cloned Code?", Proc. *IWSC*, 2011, pp. 28 – 33.

[16] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment", Proc. *MSR*, 2007, pp. 18.

[17] A. Lozano, and M. Wermelinger, "Tracking clones' imprint", Proc. *IWSC*, 2010, pp. 65 – 72.

[18] A. Lozano, and M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227 – 236.

[19] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", Proc. *ACM SAC*, 2012, pp. 1227 – 1234.

[20] M. Mondal, C. K. Roy, and K. A. Schneider, "Dispersion of Changes in Cloned and Non-cloned Code", Proc. *IWSC*, 2012, pp. 29 – 35.

[21] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider, "An Empirical Study of the Impacts of Clones in Software Maintenance", Proc. *ICPC*, 2011, pp. 242 – 245.

[22] M. Mondal, C. K. Roy, J. Krinke, and K. A. Schneider, "An Empirical Study on Clone Stability", *ACM SIGAPP ACR*, 2012, 12(3): 20 – 36.

[23] C. K. Roy, and J. R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools," Proc. *Mutation*, 2009, pp. 157 – 166.

[24] C.K. Roy, and J.R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," Proc. *ICPC*, 2008, pp. 172 – 181.

[25] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *SCP*, 2009, 74(2009): 470 – 495.

[26] C.K. Roy, and J. R. Cordy, "An Empirical Evaluation of Function Clones in Open Source Software", Proc. *WCRE*, 2008, pp. 81 – 90.

[27] C. K. Roy, and J. R. Cordy, "Scenario-based Comparison of Clone Detection Techniques", Proc. *ICPC*, 2008, pp.153 – 162.

[28] R. K. Saha, C. K. Roy, and K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies", Proc. *ICSM*, 2011, pp. 293 – 302.

[29] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study", Proc. *SCAM*, 2010, pp. 87 – 96.

[30] Shannon entropy: `http://en.wiktionary.org/wiki/Shannon_entropy#English`.

[31] "Simian - Similarity Analyser". `http://www.harukizaemon.com/simian/index.html`.

[32] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An empirical study on the maintenance of source code clones", *ESE*, 2009, 15(1): 1 – 34.

[33] Mann Whitney Wilcoxon Test: `http://elegans.som.vcu.edu/~leon/stats/utest.cgi`