Comparative Stability of Cloned and Non-cloned Code: An Empirical Study

Manishankar Mondal[1], Chanchal K. Roy[1], Md. Saidur Rahman[1], Ripon K. Saha[1], Jens Krinke[2], Kevin A. Schneider[1]
[1]Department of Computer Science, University of Saskatchewan, Canada
[2]University College London, UK
[1]{mshankar.mondal, chanchal.roy, saeed.cs, ripon.saha, kevin.schneider}@usask.ca
[2]j.krinke@ucl.ac.uk

ABSTRACT

Code cloning is a controversial software engineering practice due to contradictory claims regarding its effect on software maintenance. Code stability is a recently introduced measurement technique that has been used to determine the impact of code cloning by quantifying the changeability of a code region. Although most of the existing stability analysis studies agree that cloned code is more stable than non-cloned code, the studies have two major flaws: (i) each study only considered a single stability measurement (e.g., lines of code changed, frequency of change, age of change); and, (ii) only a small number of subject systems were analyzed and these were of limited variety.

In this paper, we present a comprehensive empirical study on code stability using three different stability measuring methods. We use a recently introduced hybrid clone detection tool, NiCAD, to detect the clones and analyze their stability in four dimensions: by clone type, by measuring method, by programming language, and by system size and age. Our four-dimensional investigation on 12 diverse subject systems written in three programming languages considering three clone types reveals that: (i) Type-1 and Type-2 clones are unstable, but Type-3 clones are not; (ii) clones in Java and C systems are not as stable as clones in C# systems; (iii) a system's development strategy might play a key role in defining its comparative code stability scenario; and, (iv) cloned and non-cloned regions of a subject system do not follow a consistent change pattern.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, Reverse Engineering and Reengineering.*

General Terms

Measurement and Experimentation

Keywords

Code Stability; Modification Frequency; Average Last Change Date; Average Age; Clone Types

1. INTRODUCTION

Frequent copy-paste activity by programmers during software development is common. Copying a code fragment from one location and pasting it to another location with or without modifications cause multiple copies of exact or closely similar code fragments to co-exist in software systems. These code fragments are known as clones. Whatever may be the reasons behind cloning, the impact of clones on software maintenance and evolution is of great concern.

The common belief is that, the presence of duplicate code poses additional challenges to software maintenance by making inconsistent changes more difficult, introducing bugs and as a result increasing maintenance efforts. From this point of view, some researchers have identified clones as "bad smells" and their studies showed that clones have negative impact on software quality and maintenance [7, 14, 15]. On the other hand, there has been a good number of empirical evidence in favour of clones concluding that clones are not harmful [1, 6, 9, 10, 18]. Instead, clones can be useful from different points of views [8].

A widely used term to assess the impact of clones on software maintenance is *stability* [6, 11, 12, 14]. Because if cloned code is more stable (changes less frequently) as compared to non-cloned code during software evolution, it can be concluded that cloned code does not significantly increase maintenance efforts. Different researchers have defined and evaluated stability from different viewpoints which can be broadly divided into two categories:

(1) Stability measurement in terms of changes: Some methodologies [6, 11, 14, 5] have measured stability by quantifying the changes to a code region using two general approaches - (i) determination of the ratio of the number of lines added, modified and deleted to the total number of lines in a code region (cloned or non-cloned) [11, 14, 5] and (ii) determination of the frequency of modifications to the cloned and non-cloned code [6] with the hypothesis that the higher the modification frequency of a code region is the less stable it is.

(2) Stability measurement in terms of age: This approach [12] determines the average last changed dates of cloned and non-cloned code of a subject system. The hypothesis is that the older the average last change date of a code region is, the more stable it is.

Table 1: Research Questions and Dimensions

	Research Questions	Dimensions
1	Do different types of clones exhibit different stability scenarios? Which type(s) of clones is (are) more vulnerable to the system's stability?	Type centric
2	Why and to what extent do the decisions made by different methods on the same subject system differ?	Method centric
3	Do different programming languages exhibit different stability scenarios?	Language centric
4	Is there any effect of system sizes and system ages on the stability of cloned and non-cloned code?	System centric

To measure the comparative stability of cloned and non-cloned code, Krinke carried out two case studies [11, 12]. In his first study, he calculated the comparative instabilities caused by cloned and non-cloned regions in terms of addition, deletion and modification in these regions whereas, in most recent study [12] (elaborated in Section 3.2), he determined the average last change dates of the regions. Both of these studies suggest cloned code to be more stable than non-cloned code. Hotta et al., in a recent study [6], calculated the modification frequencies of cloned and non-cloned code and found that the modification frequency of non-cloned code is higher than that of cloned code.

Though all of the studies [6, 11, 12, 5] generally agreed on the higher stability of cloned code over non-cloned code, we have conducted our research emphasizing on the following lackings of these studies.

(1) None of the studies was conducted considering all aspects of stability.

(2) General decisions were taken without considering a wide variety of subject systems.

(3) No study addresses the comparative stabilities as well as impacts of different clone types. This issue is important in the sense that, different types of clones might have different impacts (good or bad) on maintenance. Based on the variability of impacts, we might need to take care of clones of some specific types while leaving others without headache.

(4) Instabilities of clones on the basis of language variability were not measured. But, this information might be very important from managerial perspectives. The projects being developed using those programming languages which exhibit higher instabilities of clones might be instructed to be taken care of with more importance regarding cloning activities.

Focusing on all these issues we have performed an in-depth investigation considering three clone types using three methods including those proposed by Krinke [12] and Hotta et al.[6] in the form of four research questions listed in Table 1 from four dimensions. The third method in our investigation is our proposed variant of Krinke's method [12]. The reasons behind introducing this variant are elaborated in Section 3.3 For detecting clones, we used the recently introduced hybrid clone detection tool NiCad [3] that combines the strengths and overcomes the limitations of both text-based and AST-based clone detection techniques and exploits novel applications of a source transformation system to yield highly accurate identification of Type-1, Type-2 and Type-3 cloned code in software systems [16].

Our experimental results on three clone types of 12 subject systems written in three languages (Java, C and C#) reveal that, Type-1 and Type-2 clones are potential threats to a system's stability while Type-3 clones are not. Our language centric analysis suggests that, clones in Java and C systems exhibit higher level of instabilities as compared to those of C# systems. This is also statistically supported by the Fisher's Exact Test [4] results. We have also found that, cloned and non-cloned regions of subject systems do not follow any consistent change pattern. Moreover, the development strategy may have a strong impact on the comparative stability of cloned vs. non-cloned code.

The rest of the paper is organized as follows: Section 2 outlines the relevant research, Section 3 elaborates on the candidate methods, experimental setup is described in Section 4 and Section 5 contains the experimental results. A detailed analysis of the experimental results is presented in Section 6 while Section 7 mentions some possible validity threats of our study followed by Section 8 that contains conclusion and future directions.

2. 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. [9] introduced clone genealogy model to study clone evolution and applied the model on two medium sized Java systems. They showed that - (i) refactoring of clones may not always improve software quality and (ii) immediate refactoring of short-lived clones is not required and that such clones might not be harmful. Saha et al. [18] extended their work by extracting and evaluating code clone genealogies at the release level of 17 open source systems and reported that most of the clones do not require any refactoring effort in the maintenance phase.

Kapser and Godfrey [8] strongly argued against the conventional belief of harmfulness of clones by investigating different cloning patterns. They showed that - (i) about 71% of the cloned code has a kind of positive impact in software maintenance and (ii) cloning can be an effective way of reusing stable and mature features in software evolution.

Lozano and Wermelinger et al. performed two studies [14, 15] on the impact of clones on software maintenance considering method level granularities using CCFinder [2]. Both of their studies, though conducted on a small number of Java systems (4 in [14] and 5 in [15]), reported that clones have harmful impact on the maintenance phase because clones often increase maintenance efforts and are vulnerable to system's stability.

Juergens et al. [7] studied the impact of clones on large scale commercial systems and suggested that - (i) inconsistent changes occurs frequently with cloned code and (ii) 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 the 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. [19] in another empirical study on four subject systems reported that most of the clones are changed consistently and other inconsistently changed fragments evolve independently.

In a recent study [5] Göde et al. replicated and extended Krinke's study [11] using an incremental clone detection technique to validate the outcome of Krinke's study. He supported Krinke by assessing cloned code to be more stable than non-cloned code in general.

Code stability related research conducted by Krinke [11, 12] and Hotta et al. [6] have already been mentioned in the introduction and elaborated later in Section 3.

In our empirical study, we have replicated Krinke's [12] and Hotta et al.'s [6] methods and implemented our variant of Krinke's method [12] using NiCad [3]. Our experimental results and analysis of those results reveal inportant information about comparative stabilities and harmfulness of three clone types along with language based stability trends.

3. STABILITY MEASURING METHODS

This section discusses the three methods and associated metrics that we have implemented for our investigation. These methods follow different approaches and calculate different metrics but their aim is identical in the sense that each of these methods takes the decision about whether cloned code of a subject system is more stable than its non-cloned code. For this reason we perform a head-to-head comparison of the stability decisions taken by the metrics of these three methods and focus on the implementation and strategic differences that cause decision dissimilarities.

3.1 Modification Frequencies [6]

Hotta et al. [6] have calculated two metrics: (i) MF_d (Modification Frequencies of Duplicate code) and (ii) MF_n (Modification Frequencies of Non-Duplicate code) considering all the revisions of a given codebase extracted from SVN. Their metric calculation strategy involves identification and checking out of relevant revisions of a subject system, normalization of source files by removing blank lines, comments and indents, detection and storing of each line of duplicate code into the database. The differences between consecutive revisions are also identified and stored in the database. Then, MC_d (Modification Count in Duplicate code region) and MC_n (Modification Count in Non-Duplicate code region) are determined exploiting the information saved in the database and finally MF_d and MF_n are calculated using the following equations [6]:

$$MF_d = \frac{\sum_{r \epsilon R} MC_d(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_d(r)} \quad (1)$$

$$MF_n = \frac{\sum_{r \epsilon R} MC_n(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_n(r)} \quad (2)$$

Here, R is the number of revisions of the candidate subject system. $MC_d(r)$ and $MC_n(r)$ are the number of modifications in the duplicate and non-duplicate code regions respectively between revisions r and $(r+1)$. MF_d and MF_n are the modification frequencies of the duplicate and non-duplicate code regions of the system. $LOC(r)$ is the number of LOC in revision r. $LOC_d(r)$ and $LOC_n(r)$ are respectively the numbers of duplicate and non-duplicate LOCs in revision r.

3.2 Average Last Change Date [12]

Krinke [12] has introduced a new concept of code stability measurement by calculating the average last change dates of cloned and non-cloned regions of a codebase using the *blame* command of SVN. He considers only a single revision (generally the last revision) unlike the previous method proposed by Hotta et al. [6] that considers all the revisions up to the last one. The *blame* command on a file retrieves each line's revision and date when the line was last changed. He calculates the average last change dates of cloned and non-cloned code from the file level and system level granularities.

File level metrics: (1) Percentage of files where the average last change date of cloned code is older than that of non-cloned code (PF_c) (cloned code is older than non-cloned code) in the last revision of a subject system. **(2)** Percentage of files where the average last change date of cloned code is newer than that of non-cloned code (PF_n) (cloned code is younger than non-cloned code) in the last revision of a subject system.

System level metrics: (1) Average last change date of cloned code (ALC_c) for the last revision of a candidate subject system. **(2)** Average last change date of non-cloned code (ALC_n) for the last revision of a candidate subject system.

To calculate file level metrics in our implementation, we considered only the analyzable source files which exclude two categories of files from consideration: (i) files containing no cloned code and (ii) fully cloned files. But, system level metrics were calculated considering all source files. According to this method, the older the code is the more stable it is.

Calculation of average last change date: Suppose five lines in a file correspond to 5 revision dates (or last change dates) 01-Jan-11, 05-Jan-11, 08-Jan-11, 12-Jan-11, 20-Jan-11. The average of these dates was calculated by determining the average distance (in days) of all other dates from the oldest date 01-Jan-11. This average distance is (4+7+11+19)/4 = 10.25 and thus the average date is 10.25 days later to 01-Jan-11 yielding 11-Jan-11.

3.3 Proposed Variant of Krinke's Method

We have proposed a variant of Krinke's methodology [12] to analyze the longevity (stability) of cloned and non-cloned code by calculating their average ages. We also have used the *blame* command of SVN to calculate the age for each of the cloned and non-cloned lines in a subject system.

Suppose we have several subject systems. For a specific subject system we work on its last revision R. By applying a clone detector on revision R, we can separate the lines of each source file into two disjoint sets: (i) containing all cloned lines and (ii) containing all non-cloned lines. Different lines of a file contained in R can belong to different previous revisions. If the *blame* command on a file assigns the revision r to a line x, then we understand that line x was produced in revision r and has not been changed up to the last revision R. We denote the revision of x as $r = revision(x)$. The creation date of r is denoted as $date(r)$. In the last revision R, we can determine the age (in days) of this line by the following equation:

$$age(x) = date(R) - date(revision(x)) \quad (3)$$

We have calculated the following two average ages for cloned and non-cloned code from system level granularity.

(1) Average age of cloned code (AA_c) in the last revision of a subject system. This is calculated by considering all cloned lines of all source files of the system.

(2) Average age of non-cloned code (AA_n) in the last revision of a subject system. AA_n is calculated by considering all non-cloned lines of all source files of the system.

According to our method, a higher average age is the implication of higher stability.

We have introduced this variant emphasizing on the following issues in Krinke's method.

(1) *blame* command of SVN gives the revisions as well as revision dates of all lines of a source file including its comments and blank lines. Krinke's method does not exclude blank lines and comments from consideration. This might play a significant role on skewing the real stability scenario.

(2) As indicated in the average last change date calculation process, Krinke's method often introduces some rounding errors in its results. This might force the average last change dates of cloned and non-cloned code to be equal (There are examples in Section 5).

(3) The method's dependability on the file level metrics sometimes alters the real stability scenario. Type-3 case of 'Greenshot' is an example where both Hotta et al.'s method and our proposed variant take similar decision (non-cloned code more stable) but, file level metrics of Krinke's method alters this decision. Here, system level metrics (ALCs) of Krinke's method could not take decision because, the values of the metrics corresponding to cloned (ALC_c) and non-cloned (ALC_n) code were same.

Our proposed variant overcomes these issues while calculating stability results. It does not calculate any file level metrics because its system level metrics are adequate in decision making. This should be mentioned that, Hotta et al.'s method also ensures the exclusion of blank lines and comments from consideration through some preprocessing prior to clone detection.

3.4 Major Difference Between Hotta's Method and the Other Two Methods

Hotta et al.'s method considers *all* modifications to a region from its creation and it does not matter *when* the modifications to the region are applied. The other two methods only consider the *last* modification (which can also be the creation) and do not consider any modification before.

Suppose a file contains two lines denoted by x and y at revision 1 and this file passed through 100 commits during which x had 5 changes and y had only one change. Let the change on y have occurred at the 99th commit and the last change on x occurred at the 50th commit. A *blame* command on the last revision (100) of this file will assign x with revision 50 where y will be assigned with revision 99. According to both Krinke's method and our variant, x is older than y because the revision date corresponding to revision 50 is much older than the revision date corresponding to revision 99 and thus, x will be suggested to be more stable than y by these two methods. On the other hand, the method proposed by Hotta et al. counts the number of modifications occurred on these two lines. Consequently, Hotta et al. will suggest y to be more stable than x because the modification frequency of x will obviously be greater than that of y.

4. EXPERIMENTAL SETUP

4.1 Clone Detection

We used the NiCad [3, 17] clone detection tool to detect clones in the subject systems in our study. NiCad can detect both exact and near-miss clones at the function or block level of granularity. 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 mentioned in Table 2 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. We set the dissimilarity threshold to 20% with blind renaming of identifiers for detecting Type-3 clones. For the above settings NiCad was shown to have high precision and recall [16].

Table 2: NiCad Settings

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

Table 3: Subject Systems

	Systems	Domains	LOC	Rev
Java	DNSJava	DNS protocol	23,373	1635
	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
	Plandora	Project Management	79,853	32
C	Ctags	Code Def. Generator	33,270	774
	Tidyfornet	Wrapper for Tidy	123,409	55
	QMail Admin	Mail Management	4,054	317
	Hashkill	Password Cracker	83,269	110
C#	GreenShot	Multimedia	37,628	999
	ImgSeqScan	Multimedia	12,393	73
	Capital Resource	Database Management	75,434	122
	MonoOSC	Formats and Protocols	18,991	355

Rev = Revisions

4.2 Subject Systems

Table 3 lists the details of subject systems used in our study. We selected these subject systems because these are diverse in nature, differing in size , spanning 11 application domains, and covering three programming languages. Also, most of these systems differ from those included in the studies of Krinke[12] and Hotta et al.[6], which was intentionally done to retrieve exact stability scenarios.

5. EXPERIMENTAL RESULTS

We implemented all three candidate methods in a common framework in Java using MySQL for the backend database. Instead of using any existing implementations, we have reimplemented all these methods as we wanted to have a common framework for comparison. Values for the target metrics

Table 4: Decision Making Strategy

Method	Metrics		Decision Making	
	CC	NC	CC More Stable	NC More Stable
Hotta et al. [6]	MF_d	MF_n	$MF_d < MF_n$	$MF_n < MF_d$
Krinke [12]	ALC_c, PF_c	ALC_n, PF_n	ALC_c is older	ALC_n is older
Krinke's Variant	AA_c	AA_n	$AA_c > AA_n$	$AA_n > AA_c$

Special Decision for Krinke's Method
if $ALC_c = ALC_n$ then
$PF_c > PF_n$ implies CC more stable
$PF_n > PF_c$ implies NC more stable

CC = Cloned Code NC = Non-cloned Code

Table 6: File level metrics for two systems

Subject System	Clone Type	PF_c	PF_n
Plandora	Type-2	6	4
Greenshot	Type-3	43	12

Table 7: Modification Frequencies of Cloned (MF_d) and Non-cloned (MF_n) code by Hotta et al.'s method

		Type 1		Type 2		Type 3	
	Systems	MF_d	MF_n	MF_d	MF_n	MF_d	MF_n
Java	DNSJava	21.61	7.12	19.34	6.99	7.93	8.66
	Ant-Contrib	3.62	1.49	2.02	1.52	1.43	1.59
	Carol	8.15	6.60	4.07	3.69	9.91	8.97
	Plandora	0.44	0.92	0.45	0.97	0.55	1.11
C	Ctags	6.37	3.82	7.19	7.17	6.71	3.68
	Tidyfornet	0	5.07	0	4.87	0	4.89
	QMailAdmin	5.09	2.74	8.83	5.47	8.24	2.58
	Hash Kill	61.24	115.22	59.92	115.64	65.75	118.04
C#	GreenShot	7.94	6.07	6.92	6.07	8.13	6.06
	ImgSeqScan	0	20.93	0	21.06	0	21.29
	CapitalResource	0	67.15	0	67.31	3.63	67.11
	MonoOSC	8.58	29.14	7.92	29.23	10.62	29.63

were obtained by applying each of the methods on each of the subject systems considering three types of clones (Type-1, Type-2 and Type-3). Table 5 shows the average last change dates obtained by applying Krinke's method. Table 7 and Table 8 contain respectively the modification frequencies and average ages of cloned and non-cloned code. File level metrics for two special cases (Table 4) are shown in Table 6. Interpretation of the table data is explained below.

Almost all of the tables are self-explanatory. Decision making strategies for Tables 5, 7 and 8 are elaborated in Table 4. The stability decisions (as per Table 4) of all the metric values contained in the Tables 5, 7 and 8 are summarized in Table 9 which contains decisions for 108 (12 subject systems x 3 methods x 3 clone types) decision points corresponding to 108 cells containing decision symbols ('⊕' and '⊖', explained in the table).

For decision making regarding Krinke's method we prioritized the system level metrics (ALC_c and ALC_n) as they represent the exact scenarios of the whole system. There are only two examples of special cases as per Table 4: (i) Type-3 case of 'Greenshot' and (ii) Type-2 case of 'Plandora'. For these, the system level metrics (Table 5) are the same and so, we took decisions from file level metrics. We have given the file level metrics for these two cases in Table 6 without giving for all 36 cases (12 subject systems x 3 clone types).

6. ANALYSIS OF EXPERIMENTAL RESULTS

We presented our analysis of the experimental results from four perspectives by answering four research questions introduced in Table 1. Table 10 containing 36 (12 subject systems, 3 clone types) decision points was developed from Table 9. Each cell of this table corresponds to a decision point and implies the agreement ('⊕' or '⊖') and disagreement ('⊗') of the candidate methods in making stability decisions for that point. Meanings of '⊕', '⊖' and '⊗' are given in the table. In Table 9, the decisions of the candidate methods for Type-1 clones of 'Ctags' are same (⊖). For the Type-2 case, our proposed variant disagrees with

Table 8: Average Age in days of Cloned (AA_c) and Non-cloned (AA_n) code by the proposed variant

		Type 1		Type 2		Type 3	
	Systems	AA_c	AA_n	AA_c	AA_n	AA_c	AA_n
Java	DNSJava	2181	2441	2247	2443	2210.9	2446.9
	AntContrib	853.6	903.7	896.1	903.3	870.6	904.4
	Carol	189.6	210.9	190.3	211.3	227	209.6
	Plandora	51.82	51.32	50.6	51.4	51.5	51.32
C	Ctags	1301.4	1345.2	1351.9	1345	1564.8	1343.4
	Tidyfornet	104.5	97.9	84.9	98.1	72.8	98.3
	QMailAdmin	2664.2	2678.1	2651.7	2678.2	2644.6	2678.3
	Hash Kill	261.5	118.5	250.3	118.4	257.9	118
C#	GreenShot	103.1	97.1	102.9	97.1	94.5	97.2
	ImgSeqScan	14	20	15.6	20.3	14.4	20.4
	Capital Resource	86.7	86.5	88	86.5	89.3	86.5
	MonoOSC	315.4	313.5	347.9	313	378	312.3

Table 9: Comparative Stability Scenarios

	Methods	Krinke [12]			Hotta et al.[6]			Variant		
	Systems	*T1*	*T2*	*T3*	*T1*	*T2*	*T3*	*T1*	*T2*	*T3*
Java	DNSJava	⊖	⊖	⊖	⊖	⊖	⊕	⊖	⊖	⊖
	Ant-Contrib	⊖	⊖	⊖	⊖	⊖	⊕	⊖	⊖	⊖
	Carol	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊖	⊕
	Plandora	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊖	⊕
C	Ctags	⊖	⊖	⊕	⊖	⊖	⊖	⊖	⊕	⊕
	Tidyfornet	⊕	⊖	⊖	⊕	⊕	⊕	⊕	⊖	⊖
	QMailAdmin	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
	Hash Kill	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
C#	GreenShot	⊕	⊕	⊕	⊖	⊖	⊖	⊕	⊕	⊖
	ImgSeqScan	⊖	⊖	⊖	⊕	⊕	⊕	⊖	⊖	⊖
	CapitalResource	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
	MonoOSC	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕

⊕=*Cloned Code More Stable*
⊖=*Non-Cloned Code More Stable*
T1, T2 and T3 denote clone types 1, 2 and 3 respectively

Table 10: Overall stability decisions by methods

Lang.	**Java**				**C**				**C#**			
Clone Types	DNSJava	Ant-Contrib	Carol	Plandora	Ctags	TidyforNet	QMail Admin	Hash Kill	GreenShot	ImgSeqScan	Capital Resource	MonoOSC
Type-1	⊖	⊖	⊖	⊕	⊖	⊕	⊖	⊕	⊗	⊗	⊗	⊗
Type-2	⊖	⊖	⊖	⊗	⊗	⊗	⊖	⊕	⊗	⊗	⊕	⊕
Type-3	⊗	⊗	⊗	⊕	⊗	⊗	⊖	⊕	⊗	⊗	⊕	⊕

⊕=*All methods agree that cloned code is more stable*
⊖=*All methods agree that non-cloned code is more stable*
⊗=*Not all of the methods agreed*

Table 5: Average Last Change Dates of Cloned (ALC_c) and Non-cloned (ALC_n) code

	Clone Types / Systems	Type 1 ALC_c	Type 1 ALC_n	Type 2 ALC_c	Type 2 ALC_n	Type 3 ALC_c	Type 3 ALC_n
Java	DNSJava	24-Mar-05	26-Apr-04	21-Jan-05	24-Apr-04	31-Mar-05	19-Apr-04
	Ant-Contrib	22-Sep-06	03-Aug-06	18-Sep-06	02-Aug-06	08-Sep-06	03-Aug-06
	Carol	25-Nov-07	18-Jan-07	25-Nov-07	14-Jan-07	12-Jun-05	27-Feb-07
	Plandora	31-Jan-11	01-Feb-11	01-Feb-11	01-Feb-11	31-Jan-11	01-Feb-11
C	Ctags	27-May-07	31-Dec-06	24-Mar-07	31-Dec-06	17-Sep-06	01-Jan-07
	Tidyfornet	10-Jan-07	16-Jan-07	29-Jan-07	16-Jan-07	10-Feb-07	16-Jan-07
	QMail Admin	07-Nov-03	24-Oct-03	19-Nov-03	24-Oct-03	26-Nov-03	24-Oct-03
	Hash Kill	14-Jul-10	02-Dec-10	27-Jul-10	02-Dec-10	19-Jul-10	02-Dec-10
C#	GreenShot	11-Jun-10	21-Jun-10	12-Jun-10	21-Jun-10	20-Jun-10	20-Jun-10
	ImgSeqScan	19-Jan-11	14-Jan-11	17-Jan-11	14-Jan-11	19-Jan-11	14-Jan-11
	Capital Resource	13-Dec-08	12-Dec-08	11-Dec-08	12-Dec-08	10-Dec-08	12-Dec-08
	MonoOSC	08-Apr-09	21-Mar-09	05-Mar-09	21-Mar-09	21-Jan-09	22-Mar-09

both Krinke's and Hotta's methods, and for Type-3 clones, Hotta's method disagrees with the remaining two. So, in Table 10, Type-1 clones of 'Ctags' is marked with '$\ominus$' and both Type-2 and Type-3 clones are marked with '$\otimes$'.

6.1 Type centric analysis

In this analysis, we tried to answer the first research question (Table 1) by investigating how a particular method's decisions on a particular subject system vary with the variation of clone types. We have the following observations.

The stability decisions made by a method on a specific subject system corresponding to three clone types are similar for 24 cases (Table 9, 66.67% cases among 36 cases. Each case consists of three decisions for three clone types made by a particular method on a particular subject system) with some minor variations for the remaining cases. As an example of variations, consider the decisions made by Hotta's method for 'DNSJava'. For Type-3 case (Table 7), $MF_d < MF_n$ suggests that Type-3 clones are more stable than the corresponding non-cloned code. However, according to this method, Type-1 and Type-2 clones of 'DNSJava' are much less stable than non-cloned code (The difference of MFs for the Type-3 case is smaller compared to the differences for the other two cases).

By analyzing each clone type in Table 10 individually we have discovered some stability statistics given in Table 11 which demonstrates the following trends with the increase of clone type index from Type-1 to Type-3 -

(i) The percentage of subject systems where cloned code is more stable than non-cloned code is increasing.

(ii) The percentage of subject systems where cloned code is less stable than non-cloned code is decreasing rapidly.

(iii) For each clone type, a major amount of decision disagreements is observable and also the proportion of disagreements is increasing.

Findings of Analysis: Type-1 (exact clones) and Type-2 (clones with differences in identifier names and data types) clones are very harmful for a system because they cause the system to be more unstable than the corresponding non-cloned code. According to the agreed decisions points (Table 10) of Type-1 case -

(i) clones decrease the stability of a system with probability = No. of cells with cloned code less stable/total no. of cells = 5/12 = 0.42.

(ii) non-cloned code decreases the stability of a system with probability = 3/12 = 0.25.

For Type-2 case, these two probabilities are 0.33 (for cloned code) and 0.25 (for non-cloned code) respectively. So, for both of these cases (Type-1 and Type-2) cloned code has higher probability of decreasing the system's stability. But, the opposite is true for Type-3 case. Type-3 clones decrease stability with probability 0.08 which is much less than the probability of non-cloned code (0.33). Thus, both (i) exact copy-paste activities and (ii) cloning by renaming identifiers and changing data types should be given more attention during development as well as maintenance phase.

Table 11: Stability w.r.t. different clone types

Decision Parameters	% of Subject Systems *Type 1*	*Type 2*	*Type 3*
Cloned code more stable	25	25	33.33
Non-cloned code more stable (Cloned code less stable)	41.67	33.33	8.33
Conflicting	33.33	41.67	58.33

Table 12: Stability w.r.t. candidate methods

Decision Parameters	% of Decision Points Krinke [12]	Hotta et al.[6]	Proposed variant
Non-cloned code more stable (cloned code less stable)	55.56	44.44	52.78
Cloned code more stable	44.44	55.56	47.22

6.2 Method centric analysis

We see that Table 9 contains 108 decision points where each method contributes 36 decisions (corresponding to 12 systems and 3 clone types). From this we can retrieve the decision making scenario presented in Table 12 exhibited by the candidate methods. The table shows that Krinke's method and our variant exhibit similar statistics, but the method proposed by Hotta et al. shows a larger variation from the others. Overall, Hotta et al.'s approach suggests that cloned code is more stable while the other two suggest the opposite.

There is not only a disagreement in general, but also for the individual systems, as can be seen in Table 10. The interesting information that can be retrieved from this table is that the candidate methods have disagreements for 44.44% of the cases (16 cells of disagreements among 36 cells in Table 10). Moreover, only for two systems ('QMail Admin' and 'Hash Kill') all three methods agree on their decisions. There are some strong disagreements due to the major differences in decision strategies mentioned in Section 3.4.

Analysis of Strong Disagreements: We consider 'ImqSeqScan' as an (extreme) example. For each clone type, the method proposed by Hotta et al. shows strong disagreement to the decision of Krinke's method and our variant. Each of the three types of clones was suggested to be more stable than non-cloned code by the method proposed by Hotta et al. (Table 7). However, both Krinke's method and our variant yield the opposite decisions (Table 5 and 8). More interestingly, Hotta's method reveals that the cloned regions of 'ImgSeqScan' did not receive any change (modification frequencies of cloned code is 0, Table 7) during the entire lifetime (consisting of 73 commit transactions) where the other two methods show that the cloned code is significantly younger. In this case the regions of cloned code have only been created lately and have not been modified after creation. The following explanation will clarify this.

Suppose a subject system receives 100 commit transactions. Some clone fragments were created in some of these commits but no existing clone fragment was modified at all. In such a case, Hotta et al.'s method will see that there are no modifications in the cloned region. As a result, MF_d will be zero. On the other hand, the *blame* command will retrieve the creation dates of the clone fragments existing in the last revision of the system and Krinke's method will determine the average last change date for the cloned region considering these creation dates. If the creation dates of some clone fragments are newer than the modification dates of non-cloned fragments which forces the average last change date of the cloned region to be newer than that of the non-cloned region, Krinke's method will suggest cloned code to be less stable than non-cloned code. Thus, the cloned or non-cloned region of a subject system might be represented to be less stable than its counterpart even if it does not undergo any modifications during the entire evolution time while its counterpart does.

Finally, in answer to the second research question we can say that the stability decisions made by the candidate methods are often not similar (44.44% dissimilarities) and Hotta et al.'s method has strong disagreements with the other two methods in many cases.

Findings of Analysis: Considering all candidate methods and metrics we see that, cloned code (all three types) has higher probability to force a system into an unstable state as compared to the probability of non-cloned code. According to Table 9, cloned code is less stable than non-cloned code for 55 cells (among 108 cells). The opposite is true for the remaining cells. So the probability by which cloned code makes the system unstable is $55/108 = 0.51$ which outweighs the probability of non-cloned code (0.49). Though the difference between the probabilities is very small, it disagrees with the conclusion drawn by both Krinke [12] and Hotta et al.[6] about comparative stability. Thus, clones should be carefully maintained and refactored (if possible).

6.3 Language centric analysis

Our set of subject systems consists of four systems from each of the three languages (Java, C and C#). In Table 10, each language contributes 12 (4 subject systems, 3 clone types) decision points. Considering these decision points we have retrieved some language specific stability scenario which are presented in Table 13.

It is interesting that none of the C# systems analyzed in our study showed non-cloned code to be more stable than cloned code and also these systems exhibit the highest rate of decision dissimilarities compared to others. Java and C systems show same proportions of decision disagreements.

Table 13: Stability w.r.t. programming languages

Decision Parameters	**% of Agreed Decision Points**		
	Java	C	C#
Non cloned code more stable (Cloned code less stable)	50	33.33	0
Cloned code more stable	16.67	33.33	33.33
Conflicting decisions	33.33	33.33	66.67

Table 14: Fisher's Exact Tests for prog. languages

	Java	**C**	**Java**	**C#**	**C**	**C#**
CCLS	50	33	50	0	33	0
NCLS	17	33	17	33	33	33
DD	33	33	33	67	33	67
P = 0.0141			P <0.0001		P <0.0001	

CCLS = Cloned Code Less Stable
NCLS = Non-cloned Code Less Stable
DD = Decision Disagreements

Also, for most of the decision points regarding Java systems, cloned code was less stable than non-cloned code.

Fisher's Exact Test We performed Fisher's exact tests [4] on the three possible paired-combinations of the three languages using the values in Table 13 to see whether there are significant differences among the observed proportions of different languages. We defined the following null hypothesis. The values in Table 13 were rounded before using in Fisher's exact test.

Null Hypothesis: There is no significant difference between the stability scenarios presented by different programming languages.

From Table 14 we see that, *P value* for each paired combination of programming languages is less than 0.05. This rejects the null hypothesis and confirms that, there are significant differences among the observed scenarios of different programming languages.

Findings of Analysis: Clones in Java and C systems exhibit higher instabilities (Table 13) as compared to those of C# systems and so, developers as well as project managers should be more careful during software developments using these languages (Java and C).

6.4 System centric analysis

In the system centric analysis we have investigated whether system sizes and system ages affect the comparative stabilities by observing how modifications occur in the cloned and non-cloned regions as the system becomes older. So, we recorded and plotted the modification frequencies of four subject systems for different revisions. Such type of plotting was not possible for the other two methods because these methods work only on the last revision of a subject system. We have chosen 'DNSJava', 'Carol', 'MonoOSC' and 'Hashkill' in this investigation. 'DNSJava' and 'Carol' have large number of revisions as compared to the revision numbers of other two systems. On the other hand 'Hashkill' is much bigger than the remaining three systems in case of LOC. So, selecting these system we have covered all sorts of systems in terms of LOCs and revisions numbers covering three languages. Also, these subject systems yielded contradictory stability scenarios for Hotta et al.'s method.

For each of these systems, we plotted the modification frequencies for each of the revisions beginning from the second one for each clone type. For each revision r ($r \geq 2$), the plotted modification frequencies of cloned and non-cloned code were calculated by considering all revisions from 1 to r. The intention was to calculate the modification frequencies for r ($r \geq 2$) considering r as the current last revision.

In the graphs (not included in the paper due to space limitation), resulted from frequency plotting, we observed no consistent change pattern among the distributions of modification frequencies of the mentioned systems. So, question 4 can be answered by our observation that, system sizes and system ages do not affect the stability of cloned and non-cloned code in a consistent or correlated way.

It is worth noting that every system can have a different development strategy which can affect changes to cloned and non-cloned code. For example, programmers might be afraid of changing cloned code because of the risk of inconsistent changes and would try to restrict the changes to the non-cloned code. Another possibility is that developers are advised not to change any code of other authors, thus are forced to create a clone in order to apply a change. However, such development strategies cannot be identified by looking at the change history alone and thus it is not possible to measure their impact on cloned and non-cloned code.

7. THREATS TO VALIDITY

In the experimental setup section we mentioned the clone granularity level (block clones), difference thresholds and identifier renaming options that we have used for detecting three clone types. Different setups in corresponding clone types might result in different stability scenarios.

For some large subject systems the total numbers of revisions in the SVN repository was less than 500. This happened because these subject systems were placed under SVN control after significant amount of code was developed. This might have some effects on the code stability scenarios.

8. CONCLUSION

In this paper we presented an in depth investigation on the comparative stabilities of cloned and non-cloned code. We have shown a four dimensional analysis of our experimental results by answering four research questions.The ultimate aim of our investigation is to find out the changeabilities exhibited by different clone types and languages and whether there is any yet-undiscovered consistency in code modification biasing the stability scenarios. Our system centric analysis suggests that there is no existing biases in code modifications as well as code stabilities and system development strategy can play an important role in driving comparative stability scenarios. Our type centric analysis reveals that, Type-1 (Exact clones) and Type-2 (clones with differences in identifier names and data types) clones are very harmful for a system's stability. They exhibit higher probabilities of instabilities than the corresponding non-cloned code. So, these clone types should be given more attention both from development and management perspectives. Our method centric analysis discovers the causes of strong and weak disagreements of the candidate methodologies in taking decisions stating that the methods disagree in 44.44% of the cases. In this analysis we evaluated 108 decision points of comparative stabilities and found that, cloned code exhibits higher changeability than that of non-cloned code which opposes the already established bias ([12, 6]) about comparative stabilities of cloned vs. non-cloned code. Our language centric analysis discovers that, clones of Java and C systems show higher modification probabilities as compared to those of C# systems. This argument is also supported by statistical proof using Fisher's exact test (2 tailed). Our future plan is to perform an exhaustive empirical study for further analysis of the impacts of clones using several clone detection tools, methods and a wider range of subject systems.

Acknowledgments: This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

9. REFERENCES

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

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

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

[4] Fisher's Exact Test. `http://in-silico.net/statistics/fisher_exact_test/2x3`

[5] N. Göde, J. Harder. Clone Stability. In *Proc. CSMR*, pp. 65-74, 2011.

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

[7] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. Do Code Clones Matter? In *Proc. ICSE*, pp. 485-495, 2009.

[8] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. Emp. Soft. Eng. 13(6), pp. 645-692, 2008.

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

[10] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE*, pp. 170-178, 2007.

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

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

[13] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the Harmfulness of Cloning: A Change Based Experiment. In *Proc. MSR*, pp. 18-21, 2007.

[14] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proc. IWSC*, pp. 65-72, 2010.

[15] A. Lozano, and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. ICSM*, pp. 227-236, 2008.

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

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

[18] 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. In *Proc. SCAM*, pp. 87-96, 2010.

[19] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. In *ESE*, 15(1), pp. 1-34, 2009.