Genealogical Insights into the Facts and Fictions of Clone Removal

Minhaz F. Zibran
University of Saskatchewan
minhaz.zibran@usask.ca

Ripon K. Saha
The University of Texas at Austin
ripon@utexas.edu

Chanchal K. Roy
University of Saskatchewan
croy@cs.usask.ca

Kevin A. Schneider
University of Saskatchewan
kas@cs.usask.ca

ABSTRACT

Clone management has drawn immense interest from the research community in recent years. It is recognized that a deep understanding of how code clones change and are refactored is necessary for devising effective clone management tools and techniques. This paper presents an empirical study based on the clone genealogies from a significant number of releases of nine software systems, to characterize the patterns of clone change and removal in evolving software systems. With a blend of qualitative analysis, quantitative analysis and statistical tests of significance, we address a number of research questions. Our findings reveal insights into the removal of individual clone fragments and provide empirical evidence in support of conventional clone evolution wisdom. The results can be used to devise informed clone management tools and techniques.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, and reverse engineering*

General Terms

Experementation, Management, Measurement

Keywords

clone removal, clone evolution, refactoring, reengineering

1. INTRODUCTION

Duplicate or similar code fragments are known as code clones. Previous studies report that software systems typically contain 9%-17% [40] of code clones, and the proportion may be as high as 50% [26, 27]. Code snippets that have identical source text except for comments and layout are called *Type-1* (exact) clones. Syntactically similar code snippets, where there may be variations in the names of the identifiers/variables are known as *Type-2* clones. Code fragments that exhibit *Type-2* clone similarity but also have other differences such as added, deleted or modified statements are *Type-3* clones.

Code cloning is a popular code reuse mechanism that is used to speedup the development process and facilitate independent evolution of similar program units. However, the use of code clones may be detrimental at times. For example, copy-pasting a code fragment already containing an unknown bug may cause fault propagation. Moreover, during the maintenance phase, a change in a clone fragment may necessitate consistent changes in all of its cloned fragments, and any inconsistencies may introduce vulnerabilities [41, 42, 43]. Thus, code clones may have a significant impact on the development and maintenance of software systems.

Despite ongoing research on the positive and negative effects of code clones [11, 12, 15, 21, 24, 25, 36], researchers and practitioners have come to an accord for the need of active and informed clone management [44, 45] including documentation and removal of clones through refactoring. However, code clones can often be desirable, and aggressive removal of clones through refactoring may not be a good idea [15, 41, 42, 43], given the risks and efforts involved in such activities. In this regard, a number of classification schemes [2, 13, 16, 33], metric based selection approaches [1, 4, 10], and an effort model [41, 42, 43] have been proposed to identify potential clones for refactoring. Still, for many systems, clone management and removal is yet to be a part of the daily maintenance activities [8]. Despite more than a decade of software clone research, clone management remains far from industrial adoption, and this area has gained more focus from the community in recent years [47].

A deep understanding of how individual clones change during their evolution, and which criteria cause their removal from the system, can help in devising effective strategies and tool support for clone management. A number of studies on near-miss clone evolution [30, 32, 40, 46] are found in the literature, which attempt to inform clone management [39, 44, 47]. These studies on clone evolution and programmers' psychology lead to some common beliefs and at times even contradictions about the traits of clone evolution. For example, the study of Kim et al. [15] suggests that many clones are *volatile* (i.e., disappear shortly after they are created), while the study of Lozano and Wermelinger [18] suggests otherwise.

This paper focuses on the patterns of changes and removal of code clones during the evolution of software systems. In particular, we formulate the following eight research questions to capture different characteristics of clone change and removal. Some of the research questions correspond to common beliefs (or, contradictions) in the community; but we want to develop empirical evidence based on a systematic genealogy-based study on clone change and removal in evolving software systems.

RQ1: *Do the sizes of the groups of clones make any difference in clone removal in practice?* — Kim et al. [14] suspected that frequently copied code fragments (i.e., larger clone-groups) can be good candidates for clone refactoring.

RQ2: *Do the sizes of the individual clone fragments in terms of the number of lines impact clone removal in practice?* — Larger clone fragments can be attractive candidates for refactoring, as conjectured by Kim et al. [14].

RQ3: *For a group of clones, does the distribution of the clones in the file system hierarchy impact their removal in practice?* — Göde [8] reported that the developers were more interested in refactoring closely located clones.

RQ4: *Is there any relationship between any particular type of changes in the clones and their removal?* — This is still an open question, as far as we are concerned. If there exists any relationship between a particular type of changes and clone removal, the clone management tools can focus on supporting that category of changes.

RQ5: *How frequently do the clones experience changes before they are removed from the system?* — There is an ongoing debate on the stability of code clones [9, 17, 20].

RQ6: *Does the granularity (entire function bodies or syntactic blocks) of clones make any difference in their removal in practice?* — A recent study of Göde [8] reported many instances of removal of block clones by *extract method* refactoring.

RQ7: *Does the textual similarity in the source code of the clones have any effect in the removal of clones in practice?* — Very similar (e.g., *Type-1*) clones can be expected to be easier to refactor than very dissimilar (e.g., *Type-3*) clones.

RQ8: *During the evolution of the software systems, when does clone removal take place?* — This question addresses the aforementioned contradiction about the volatility of clones.

To address the research questions, we carry out a systematic study based on code clone *genealogy* [15, 31], which maps the individual clone fragments across subsequent releases over their evolution. We investigate the changes and removal of individual clones in 329 releases of nine diverse open-source software systems written in Java, C, and C#. Then we analyze them against a wide range of metrics and characterization criteria. In the light of a combination of qualitative analysis, quantitative analysis and statistical tests of significance, we derive the answers to the research questions.

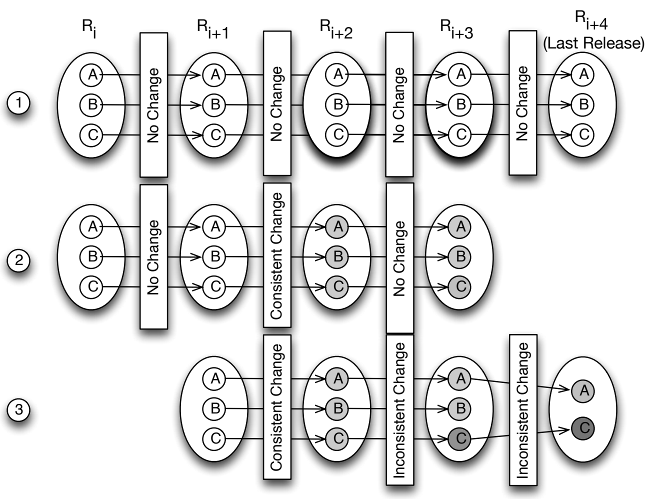


Figure 1. Different types of clone genealogies

We believe, such an empirical study on the characteristics of changes and removal of individual near-miss clone fragments is timely and addresses a gap in the literature. Our study is based on genealogies of near-miss clones including not only *Type-1* and *Type-2* clones, but also *Type-3* clones. This work is an extension to our previous work [46], which was the first genealogy-based study on the evolution, changes, and removal of near-miss code clones including *Type-3*. In this work, we significantly extend our previous study by including 101 more releases of three additional subject systems, an additional research question, and more in-depth analysis.

The rest of this paper is organized as follows. In Section 2, we introduce the terminology and metrics used in our study. In Section 3, we describe the setup and procedure of our empirical study. Section 4 presents the findings our study. In Section 5, we discuss the possible threats to the validity of our study. Section 6 accommodates related work, and Section 7 concludes the paper.

2. TERMINOLOGY AND METRICS

In this section, we introduce the terminology and metrics used in this paper to characterize the changes and removal of code clones. Some of the metrics and and criteria are adopted from earlier studies found in the literature [3, 7, 8, 15].

Clone Genealogy: A set of clone fragments that are clones of each other form a *clone-group*. A *clone genealogy* refers to a set of one or more lineage(s) originating from the same clone-group, whereas, a *clone lineage* is a sequence of clone-groups evolving over a series of releases of the software system. Figure 1 shows several examples.

Consistent and Inconsistent Change: If all clones in the clone-group experience the same set of changes during the transition between releases, then such changes are characterized as being a *consistent change*, otherwise the changes are regarded as being *inconsistent*.

Table 1. Software systems subject to our empirical study

Prog. Lang.	Subject System	No. of Releases	Releases Start	Releases End	Dates (mm/dd/yy) Start	Dates (mm/dd/yy) End	Duration (months)	Source Lines of Code (SLOC ranges)
Java	dnsjava	50	0.9.2	2.1.1	04/19/99	02/10/11	131	6,290 – 15,018
	JabRef	27	1.5	2.4.2	08/15/04	11/01/08	50	22,041 – 69,170
	ArgoUML	48	0.27.1	0.32.beta2	10/04/08	01/24/11	26	176,618 – 202,555
C	ZABBIX	31	1.0	1.8.4	03/23/04	06/01/11	86	9,252 – 62,845
	Conky	28	1.1	1.8.1	06/20/05	10/05/10	62	6,555 – 39,810
	Claws Mail	44	2.0.0	3.7.9	06/30/06	04/09/11	63	1,33,642 – 1,89,786
C#	CruiseControl	31	0.7.rc1	1.8.4	11/08/04	09/01/13	98	35,895 – 1,82,032
	iTextSharp	22	5.0.0	5.4.4	12/08/09	09/16/13	45	1,72,573 – 2,17,328
	ZedGraph	48	1.1	5.1.5	08/02/04	12/12/08	52	2,439 – 26,433

Consistently Changed Clone-Group: If the genealogy of a clone-group has any consistent change pattern(s) but does not have any inconsistent change patterns during evolution, it is classified as a *consistently changed* clone-group. The clone-group associated with the second genealogy in Figure 1 is an example of a consistently changed clone-group as there is a consistent change between releases R_{i+1} and R_{i+2}.

Inconsistently Changed Clone-Group: If the genealogy of a clone-group has any inconsistent change pattern(s) throughout the entire evolution period, it is characterized as an *inconsistently changed* clone-group. The clone-group associated with the third genealogy in Figure 1 is an inconsistently changed clone-group as there is an inconsistent change between releases R_{i+2} and R_{i+3}.

Static, Alive, Dead Clone-Group: Static clone-groups are those which propagate through subsequent releases having no textual change in the clones. A clone-group is called *dead* if it disappears before reaching the final release under consideration, otherwise the clone-group is considered *alive.* The clone-groups associated with the first, second and third genealogies in Figure 1 represents static, dead, and alive clone-groups respectively.

Textual Similarity: The textual similarity between two code snippets S_1 and S_2, denoted by $\int(S_1, S_2)$, is determined by calculating the identical lines with respect to their sizes, as defined by the following formula[1].

$$\int(S_1, S_2) = \frac{2 \times |\ell_1 \cap \ell_2|}{|\ell_1| + |\ell_2|} \tag{1}$$

where ℓ_1 and ℓ_2 are the ordered sets of pretty-printed lines in S_1 and S_2 respectively. $|\ell_1 \cap \ell_2|$ is the number of common ordered lines between ℓ_1 and ℓ_2, calculated using the longest common subsequence (LCS) algorithm. The textual similarity of a clone-group G, denoted as $\int(G)$ is the average of the textual similarities between all clone pairs in that group. Mathematically,

$$\int(G) = \frac{\sum\limits_{S_i, S_j \in G} \int(S_i, S_j)}{\binom{|G|}{2}} \tag{2}$$

[1]In the area of Information Retrieval, this similarity measurement is known as the *Dice Coefficient*.

Entropy of Dispersion: We used an entropy measure to characterize the file level physical distribution of the clones in a clone-group. Such an entropy measurement, sometimes referred to as *Shannon entropy*, is commonly used in the area of Information Theory. In this work, the entropy of dispersion of the clones in clone-group G is calculated using Equation 3 as follows:

$$\mathit{entropy}(G) = \sum_{i \in \mathcal{F}_G} -p_i \log(p_i) \tag{3}$$

where, $\mathcal{F}_G$ denotes the set of distinct files hosting the clones in clone-group G, and p_i denotes the probability of the clones being located in file i.

For example, if all the clone fragments reside in the same file, the dispersion entropy will be 0.0. If the entropy is low, clones are densely located in only a few files. If the entropy is high, the clones are scattered across different files.

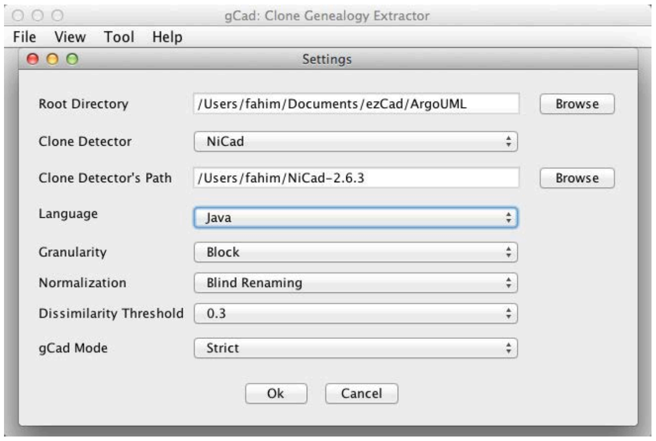
3. STUDY SETUP

To investigate the research questions outlined in Section 1, we study the clone genealogies across releases of nine diverse open-source software systems (Table 1) written in Java, C, and C#.

In the selection of the subject systems, we followed a number of criteria. First, we tried to include software systems that had reasonably large sizes and large number of releases. In computation of a system's size, we took into account only the source code lines (SLOC) written in the particular programming language that the software system is categorized in Table 1. We excluded comments, blank lines, and lines of code written in any other programming languages. Second, in our study, we tried to include subject systems from diverse application domains. Third, we preferred those open-source software systems, which were used in earlier studies [15, 30, 31, 40] reported in the literature.

3.1 Extraction of Genealogies

For the extraction of clone genealogies, we used an extended version of `gCad` [31] clone genealogy extractor that we developed. `gCad` can construct and classify genealogies of all three types (*Type-1, Type-2*, and *Type-3*) of clones that we are interested in. Details of how `gCad` operates and computes clone genealogies can be found elsewhere [31]. As per



Root Directory /Users/fahim/Documents/ezCad/ArgoUML Browse
Clone Detector NiCad
Clone Detector's Path /Users/fahim/NiCad-2.6.3 Browse
Language Java
Granularity Block
Normalization Blind Renaming
Dissimilarity Threshold 0.3
gCad Mode Strict
Ok Cancel

Figure 2. gCad settings for genealogy extraction

the need of this study, we significantly extended and customized the tool with a carefully designed graphical user interface (GUI), and a set of appropriate features to compute the necessary metrics. For the purpose of our study, we carefully chose a set of gCad's configuration parameters as shown in Figure 2.

For the detection of code clones, we selected the NiCad-2.6.3 [5, 6, 29] clone detector, which is a state-of-the-art clone detection tool reported to be effective in detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones with high precision and recall [28, 29, 34]. gCad invokes NiCad to separately detect clones from every release of each of the subject systems. In invoking the clone detector, some of the gCad parameters are passed to NiCad to guide the process of detecting *Type-1, Type-2,* and *Type-3* clones at the chosen granularity (syntactic blocks for our study) and dissimilarity threshold (0.3 for our study).

The dissimilarity threshold (Figure 2) is a size-sensitive dissimilarity threshold that plays a vital role in guiding NiCad in the detection of *Type-3* clones. For our study, the dissimilarity threshold was set to 0.3, which signifies that NiCad detects two code fragments as clones if at least 70% of their pretty-printed text lines are the same (i.e., if at most 30% lines are different). The normalization option "blind-renaming" tells NiCad to ignore the differences in the names of identifiers/variables, and thus it is a significant parameter for the detection of *Type-2* clones.

From the clone detection results obtained from NiCad, for each of the subject systems, we separately constructed the clone genealogies using gCad. We operated gCad in 'strict' mode to construct and characterize clone genealogies. In 'strict' mode, gCad captures and takes into account all types of changes in the source code lines of clone-pairs, irrespective of whether those changes took place in their corresponding similar or dissimilar lines of code. Details of how gCad operates in different modes can be found elsewhere [31].

3.2 Investigation

We examined all the dead genealogies to see how the clones were removed. We also examined how the individual clone fragments changed during their evolution over a series of releases. Since, the inconsistent changes to clones are believed to be a common phenomena that produce vulnerabilities in a system [40, 41, 43], we characterized the clone changes as consistent versus inconsistent. In addition, we captured how frequently a clone-group changes during the evolution before its removal. For quantitative analysis, we computed the necessary metrics according to the categorization described in Section 2.

4. FINDINGS

The findings of our study are derived from qualitative and quantitative analyses of the changes and removal of individual clone fragments. We also apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [19] with $\alpha = 0.05$, to determine the statistical significance of the findings. Using the *Shapiro-Wilk* test [19] and *Q-Q* plot [19], we examined the distribution of the data, and found that some of the observations exhibited normal distributions while some others did not. Therefore, we chose to use the *MWW* non-parametric test, which does not assume the normal distribution of the data, and thus, is appropriate for data that exhibit or do not exhibit normal distribution.

4.1 Size of the Clone-Groups

To capture the relationship between the number of fragments in a clone-group and clone removal, we computed the average number of fragments in the removed clone-groups and that of the alive clone-groups for each of the subject systems (Table 2). As seen in Table 2, for all the subject systems, the average size of the alive clone-groups is higher than those of the removed clone-groups. During our manual investigation, we found that the developers refactored clone-groups that had only two or three clone fragments. Similarly, we found that in JabRef, there were 74 clone-groups having more than three fragments, and only four of them were refactored. This gives the impression that developers are more inclined to remove smaller clone-groups. To statistically verify this, we address the second research question *RQ1*, and formulate our null hypothesis as follows.

H_0^1: *The size of a clone group does NOT make a difference in clone removal in practice.*

A *MWW* test ($P = 0.35$) *fails to reject* (as, $P > \alpha$) the null hypothesis, which implies that the difference is not statistically significant. Hence, we answer the research question *RQ1* as follows.

Ans. to RQ1: *The size of the clone-groups (in terms of the number of member clone fragments) does not make a statistically significant difference in clone removal in practice.*

Although, in our study, the sizes of the removed clone-groups (in terms of the number of clone fragments) appears to be consistently lower than the alive clone-groups, this might have happened simply by chance in the software systems in our study. A larger study with many software systems may be required to further investigate the possibility of statistical significance of the pattern we found between the sizes of the clone-groups and their removal.

Table 2. Sizes of removed and alive clone-groups

Prog. Lang.	Subject System	Avg. Sizes of Clone-groups	
		Removed	Alive
Java	dnsjava	2.25	2.75
	JabRef	2.31	4.17
	ArgoUML	2.12	9.12
C	ZABBIX	2.31	4.53
	Conky	2.37	9.31
	Claws Mail	2.88	2.95
C#	CruiseControl	2.32	3.58
	iTextSharp	2.21	5.80
	ZedGraph	2.15	2.50

Table 3. Average sizes (SLOC) of clone fragments

Subject System	Removed		Alive	
	Average	SD	Average	SD
dnsjava	10.00	3.00	11.00	5.00
JabRef	17.00	15.00	13.00	9.00
ArgoUML	16.00	13.00	15.00	19.00
ZABBIX	26.00	25.00	21.00	21.00
Conky	20.00	23.00	15.00	7.00
ClawsMail	15.00	9.00	16.00	18.00
CruiseControl	8.85	4.50	9.46	5.61
iTextSharp	13.99	13.62	10.76	11.00
ZedGraph	15.70	15.45	12.34	12.67

SD = Standard Deviation

4.2 Size of the Clone Fragments

The sizes of the clone fragments can be expected to have a relationship with the refactoring effort, especially when the candidate clone-group includes near-miss (*Type-2* and *Type-3*) clones beyond *Type-1*.

To examine the relationship between clone removal and the SLOC per clone fragment in the clone-groups, we separately computed the average number of pretty-printed SLOC per fragment for the removed clones as well as for the alive clones. We also calculated the standard deviations for each of the measurements to capture the degree of variations. The results are presented in Table 3.

As can be observed from Table 3, there are subtle differences in the sizes of the clone fragments of both the removed and alive clone-groups. For six of the nine subject systems (`ZabRef`, `ArgoUML`, `ZABBIX`, `Conky`, `iTextSharp`, and `ZedGraph`), the average sizes of clone fragments of removed clone-groups appear to be significantly higher than those of the alive clone-groups. Hence, the anticipation of Kim et al. [14] saying – developers are more interested in getting rid of larger clones – appears to be true.

Addressing the research question *RQ2*, we now formulate our null hypothesis as follows.

H_0^2**:** *The size of the individual clones in terms of number of lines does NOT impact clone removal in practice.*

A *MWW* test ($P = 0.001$) over the series of sizes for the removed and alive clones *rejects* (as, $P < \alpha$) the null hypothesis. From the analysis described above, we answer research question *RQ2* as follows.

Ans. to RQ2: *The size of the individual clones in terms of number of lines does have a statistically significant impact on clone removal in practice, and larger clone fragments appear to be attractive for removal in practice.*

4.3 Entropy of Dispersion

In Table 4, we present the entropy of dispersion of both the removed and alive clones for all the subject systems. From the developer's perspective, refactoring/removal of co-located clones may require less effort than that needed for refactoring clones scattered over the code base. This can be expected to hold true due to several reasons. In the refactoring of scattered clones the developers might need to spend much time and effort to navigate to, understand the contexts, and make careful modifications at different locations of the code base.

In Table 4, we see that for each of the subject systems, the average entropy of dispersion for the removed clones is much lower than that for the alive clones. This indicates those clone-groups whose member clone fragments are closely located in the code base are relatively more attractive for refactoring/removal. To determine whether the initial observation *significantly* supports the expectation, we again conducted a *MWW* test with the null hypothesis as follows.

H_0^3**:** *For a group of clones, the distribution of individual clones in the file system hierarchy does NOT impact their removal in practice.*

The hypothesis addresses the research question *RQ3*. A *MWW* test ($P = 0.233$) between the entropy values for both the removed and alive clones (over all the systems) *fails to reject* (as, $P > \alpha$) the null hypothesis. This implies that there exists no relationship between the entropy of dispersion and clone removal in practice. Therefore, we derive the answer to research question *RQ3* as follows.

Ans. to RQ3: *For a group of clones, the distribution of individual clones in the file system hierarchy does not have a statistically significant impact on their removal in practice.*

As we delved deeper through manual investigation, we found a strange phenomenon in the relationship between entropy and the number of clone fragments that were removed. Most of the removed clone-groups had two fragments, if their entropy was greater than zero, i.e., they were not really located in the same file. For example, in `JabRef` and `ZABBIX`, developers refactored 37 and 43 clone-groups respectively, all of which had entropy higher than zero. Among them only two clone-groups in JabRef and 10 clone-groups in `ZABBIX` had three clone fragments, while the rest had only two fragments.

4.4 Change Patterns

Despite the realized advantages of code cloning, it is also true that code clones may have a significant impact on software development and maintenance in several ways. First,

Table 5. Removal of clone-groups classified by change patterns

Prog. Lang.	Subject System	Static Clone-Groups Total	Static Clone-Groups Removed	Static Clone-Groups [%]	Consistently Changed CG Total	Consistently Changed CG Removed	Consistently Changed CG [%]	Inconsistently Changed CG Total	Inconsistently Changed CG Removed	Inconsistently Changed CG [%]
Java	dnsjava	60	27	45.00	8	3	37.50	49	27	55.10
	JabRef	217	52	23.96	53	3	5.66	132	15	11.36
	ArgoUML	1435	109	7.60	39	4	10.26	440	19	4.31
C	ZABBIX	166	88	53.01	61	18	29.51	109	35	32.11
	Conky	121	44	36.36	19	7	36.84	37	16	43.24
	ClawsMail	445	58	13.03	172	7	4.07	304	7	2.30
C#	CruiseControl	528	187	35.41	119	35	29.41	388	133	34.27
	iTextSharp	1259	99	7.86	103	8	7.76	325	66	20.31
	ZedGraph	225	161	74.22	33	12	36.36	79	45	56.96

Table 4. Comparison of entropy of dispersion

Prog. Lang.	Subject System	Clones Removed	Clones Alive
Java	dnsjava	0.71	0.90
	JabRef	0.53	0.98
	ArgoUML	0.82	1.30
C	ZABBIX	0.35	0.53
	Conky	0.18	0.24
	Claws Mail	0.30	0.70
C#	CruiseControl	0.18	0.23
	iTextSharp	0.22	0.38
	ZedGraph	0.17	0.10

the reuse by copy-pasting of any code segment that already contains unknown faults, results in propagation of those faults to all the copies. Second, when a change is made in a code fragment, consistent changes are often expected in all its clone fragments, while any inconsistencies may introduce new faults. Third, if a bug is found in a certain code fragment, there remains a possibility that similar bugs can be found in the clones of the fragment, and thus may necessitate consistent propagation of that bug-fix to all the clones.

Thus, whether the clones changed consistently, inconsistently, or remained static during the evolution of a software system, may have implications in clone management in future releases. Therefore, we categorized the clones based on whether they remained unchanged, or changed consistently or inconsistently, and what percentage of such clones were actually removed during the evolution of the system. For each of the systems, the total number of clones of each of these three categories and the percentage of them that were removed, are presented in Table 5.

As we can see in Table 5, for each of the subject system, the number of static clone-groups is the highest while the number of the consistently changed clone-groups is the lowest. To examine any trends in the existence of static, consistently changed, and inconsistently changed clone-groups in the systems, we again conducted *MWW* tests between each two of the three categories of changes (total number) occurred in the clone-groups over all the systems. The results

Table 6. *MWW* tests over of categories of changes

Change Types	No Change	Consistent Change	Inconsistent Change
No Change	-	$P = 0.003$	$P = 0.158$
Consistent Change	$P = 0.003$	-	$P = 0.042$
Inconsistent Change	$P = 0.158$	$P = 0.042$	-

Table 7. *MWW* tests over removal of clones

Clone Categories	Static	Consistently Changed	Inconsistently Changed
Static	-	$P = 0.31$	$P = 0.695$
Consistently Changed	$P = 0.31$	-	$P = 0.536$
Inconsistently Changed	$P = 0.695$	$P = 0.536$	-

of the *MWW* tests are presented in Table 6, which suggest significant difference in occurrence of the three categories of changes (as, $P < \alpha$), except that the difference in the number of inconsistent changes clones and no-changes are found to be statistically insignificant (as, $P > \alpha$).

With respect to clone removal, from Table 5, we see that for six of the nine systems (`JabRef`, `ZABBIX`, `ClawsMail`, `CruiseControl`, `iTextSharp`, and `ZedGraph`), the majority of the removed clones are static clone-groups. The removal of inconsistently changed clone-groups were found to occur most often in two of the systems (`dnsjava` and `Conky`), whereas, the removal of consistently changed clones dominated in `ArgoUML`.

A high-level perception from the results in the table may indicate that the static clone-groups can be more susceptible to removal. To verify such an observation, we carried out *MWW* tests between each pair of the three categories of clone *removal* over all the systems. The results of the *MWW* tests, as presented in Table 7, also suggest that there is no significant difference in the removal of static, consistently changed and inconsistently changed clone-groups (as, $P > \alpha$ in all cases).

Table 8. Frequency of changes before removal

Prog. Lang.	Subject System	Change Frequency 1	2	>2	Average
Java	dnsjava	16	9	5	1.80
	JabRef	11	4	3	1.72
	ArgoUML	17	4	2	1.48
C	ZABBIX	30	16	7	1.74
	Conky	10	8	5	1.95
	ClawsMail	9	2	5	1.57
C#	CruiseControl	103	45	20	0.75
	iTextSharp	62	11	1	0.50
	ZedGraph	40	12	5	0.39

These observations lead to the answer to the research question *RQ4* as follows.

Ans. to RQ4: *The majority of the clones do not experience any changes during their evolution. Those clones that experience changes, majority of those clone-groups undergo inconsistent changes. However, there is no statistically significant relationship between any particular type of changes in the clones, and their removal at a later release.*

4.5 Frequency of Changes

The frequency of changes to the clone-groups is an important criterion in clone management, since changing source code can be expensive, while making consistent changes to clones may involve significant effort and risks. Indeed, the modifications of a clone fragment needing effort, and the required effort can be multiplied by the size of the corresponding clone-group, to make consistent changes to all clone fragments in the clone-group. This is one of the areas where clone management tool support may contribute by facilitating clone merging, or consistent change propagation.

Thus, we examined how frequently the clone-groups underwent changes before their removal. In Table 8, we present the number of clone-groups that, before removal, underwent changes only once, twice, and more than twice. As seen in the table, most of the removed clones were changed only once. For the clone-groups that changed at least once, their average change frequency is less than two, over all the subject systems. From our manual verification, we found that very few clone-groups underwent changes more than twice before their removal. On the other hand, we also found many clone-groups remained alive although they experienced minor or significant changes. However, we confined our focus to the changes of the removed clone-groups to get a complete picture over the entire life-time of the clone-groups. Now, we derive the answer to the research question *RQ5* as follows.

Ans. to RQ5: *Most clones do not undergo frequent changes before their removal.*

4.6 Level of Granularity

The *extract method* refactoring pattern is perhaps the most highlighted technique for removing clones at the granularity of syntactic blocks. Thus, we may expect evidence of many instances of block clone removal. Alternatively, functions typically contain a somewhat complete implementation of certain features or program logic and so it may be easier to remove/refactor clones at the granularity of entire function bodies, rather than at the granularity of smaller syntactic blocks.

To determine whether there exist any relationships between clone removal and clone granularity, we examined both levels of granularities – function/method and syntactic block. Note that the body of a function also constitutes a block. Therefore, we distinguish *true* functions clones from the *true* block clones. A true function clone fragment spans the entire body of a function, whereas a true block clone must not constitute the entire body of a function.

Extended `gCad` is capable of differentiating true function clones from the true block clones. Any clone-group that is composed of only true function clones is categorized as a group of function clones, whereas, clone-groups consisting of only true block clones are categorized as groups of block clones. Separate genealogies are constructed for the clones at these two levels of granularity.

Over all releases of each of the subject systems, the total number and proportions of both the groups of function clones versus the block clones are presented in Table 9. The clone detection results for each of the systems identified clone-groups that contained both true function clones and true block clones. Therefore, it is not possible to categorize such a group as a group of only true function clones or only true block clones. This is why the total number of clone-groups reported in Table 9 is lower than that of Table 5. Addressing the research question *RQ*6, we now formulate our null hypothesis as follows.

H_0^6**:** *The granularity (entire function bodies or syntactic blocks) of clones does NOT make any difference in their removal in practice.*

A *MWW* test ($P = 0.93$) over the proportions of the removal of both true function and block clones *fails to reject* (as, $P > \alpha$) the null hypothesis.

Table 9 shows that developers remove both function and block clones as per their needs, as we do not see significant differences between the proportions of removal of function clones and block clones. For `ZABBIX` and `Conky`, the proportion of block clones removal is slightly higher. It seems that the clone removal rates for the two larger systems, `ArgoUML` and `Claws Mail` are far lower than the smaller systems. On the other hand, it appears that the developers of the relatively small systems `dnsjava`, `ZABBIX`, and `Conky` were more aware of the clones and were active in removing them through refactoring. From manual investigation, we found only one and two *Type-1* function clones in `dnsjava` and `Conky` respectively. Though as many as eight *Type-1* function clones were found in `ZABBIX`, seven of them were removed during the evolution of the system. Based on the above discussion, we now derive the answer to the research question *RQ6* as follows.

Ans. to RQ6: *In practice, the granularity (entire function bodies or syntactic blocks) of clones does not make any statistically significant difference in their removal.*

Table 9. Removal of clone-groups at the granularities of function and block

Prog. Lang.	Subject System	Function Clones Total	Function Clones Removed	Function Clones [%]	Block Clones Total	Block Clones Removed	Block Clones [%]
Java	dnsjava	69	37	53.62	25	15	60.00
	JabRef	204	41	20.09	110	21	19.09
	ArgoUML	1183	97	8.19	305	20	6.55
C	ZABBIX	201	78	38.80	134	62	46.26
	Conky	115	35	30.43	59	30	50.84
	Claws Mail	510	40	7.84	337	29	8.60
C#	CruiseControl	889	354	39.82	1032	355	34.40
	iTextSharp	999	186	18.62	1687	173	10.25
	ZedGraph	229	162	70.74	337	218	64.69

Table 10. Actual and normalized textual similarity of removed and alive clone-groups

		Actual Textual Similarity				Normalized Textual Similarity			
Prog.	Subject	Removed Clones		Alive Clones		Removed Clones		Alive Clones	
Lang.	System	Average	SD	Average	SD	Average	SD	Average	SD
Java	dnsjava	0.60	0.20	0.67	0.18	0.80	0.12	0.81	0.10
	JabRef	0.76	0.18	0.68	0.18	0.85	0.13	0.82	0.11
	ArgoUML	0.76	0.20	0.66	0.17	0.85	0.14	0.80	0.14
C	ZABBIX	0.72	0.19	0.73	0.17	0.83	0.16	0.83	0.11
	Conky	0.76	0.16	0.69	0.15	0.88	0.09	0.84	0.09
	Claws Mail	0.73	0.20	0.65	0.22	0.87	0.12	0.82	0.15
C#	CruiseControl	0.67	0.19	0.68	0.18	0.83	0.09	0.84	0.09
	iTextSharp	0.72	0.22	0.66	0.19	0.86	0.11	0.84	0.12
	ZedGraph	0.80	0.22	0.70	0.22	0.91	0.14	0.87	0.14

SD = Standard Deviation

4.7 Textual Similarity

In Table 10, we present the average text similarities (actual and normalized) of the removed clones and the alive clones for each of the subject systems. Indeed, the degree of textual similarity among the clone fragments in a clone-group is important information as it corresponds to the differences between the clone fragments. Refactoring a clone-group with many variations can require more effort than refactoring a group of identical or very similar clones. Thus, the textual similarity for the clones in a clone-group can be expected to be proportional to the necessary efforts for refactoring them. Taking these into consideration, we address the research question *RQ1*, and formulate our null hypothesis as follows:

H_0^{7a}**:** *Clone removal by the developers is NOT dictated by the similarity of program text (without normalization) in the clone fragments.*

From the table, we can see that the average *actual* textual similarity of removed clones for four systems (`JabRef, ArgoUML, Conky`, and `Claws Mail`) is higher than that of alive clones, while the other two subject systems (`dnsjava` and `ZABBIX`) exhibit slightly the opposite trend. A *MWW test* ($P = 0.041$), on the data of actual textual similarity in the alive and removed clones, *rejects* (as, $P < \alpha$) the null hypothesis H_0^{7a}, which indicates statistically significant relationship between the textual similarity of clones and their removal.

Sometimes the actual textual similarity does not estimate the actual effort for refactoring. For example, in case of different identifiers names in different clone fragments, textual similarity of a clone-group may be very low although they are easily refactorable, especially when there is refactoring support from the IDEs (Integrated Development Environments). That is why we also investigated the normalized textual similarity by removing the identifier differences. If we look at the normalized text similarities of removed and alive clones, we again see that the average *normalized* textual similarity for the removed clones is slightly higher than the alive clones in those four systems. The trend is slightly the opposite for `dnsjava`, while for `ZABBIX`, both the removed and alive clones exhibit equal average normalized textual similarity. With respect to the normalized text similarities between the remove and alive clones, we formulate another hypothesis as follows:

H_0^{7b}**:** *Clone removal by the developers is NOT influenced by the similarity of* ***normalized*** *program text in the clone fragments.*

A *MWW* test ($P = 0.091$) *fails to reject* (as, $P > \alpha$) the null hypothesis H_0^{7b}, suggesting that there is *no statistically significant difference* in the normalized text similarities between the removed and alive clones. However, the P value in the case of normalized textual similarity is much lower than that of the actual textual similarity, and this hints that there might be some influence of the differences in the names of variables/identifiers over clone removal.

Combining the observations for the actual and normalized text similarities over the removed and alive clones, we can now derive the answer to the research question *RQ7* as follows.

Ans. to RQ7: *The textual similarity in the source code of the clones does have statistically significant effect in the removal of clones, and the differences in the names of the variable/identifiers play the major role in this regard.*

This finding indicates that *Type-1* clones are most attractive to the developer for refactoring, and the developers, in practice, are more inclined in refactoring *Type-2* clones than refactoring *Type-3*.

4.8 Age

The information about the age (in terms of the number of releases the clone-groups remain alive before removal) of clone genealogies can indicate how quickly the developers act to remove clones. In order to examine this phenomenon, for each of the systems, we computed the age of each clone-group that was removed in any of the subsequent releases.

In Figure 3, we present the proportion of clone-groups found to have been removed in a subsequent releases. As the figures (Figure 2(a), Figure 2(b), and Figure 2(c)) show, majority of the dead clones in five of the subject systems (`ArgoUML, JabRef, ZABBIX, Conky,` and `ZedGraph`) were removed within the initial five to ten releases. This observation is consistent with that reported by Kim et al. [15], suggesting that many of the clones are possibly *volatile*.

However, in all the systems, a good number clones remained alive over a long sequence of releases before their removal. For example, 17% of the refactored clone-groups in `ArgoUML` remained alive in 43 subsequent releases, while 35% of the clone-groups in `Claws Mail` propagated over 27 subsequent releases, before their removal. Similar trends were found in other systems as well. From the above discussion, we answer the research question *RQ8* as follows.

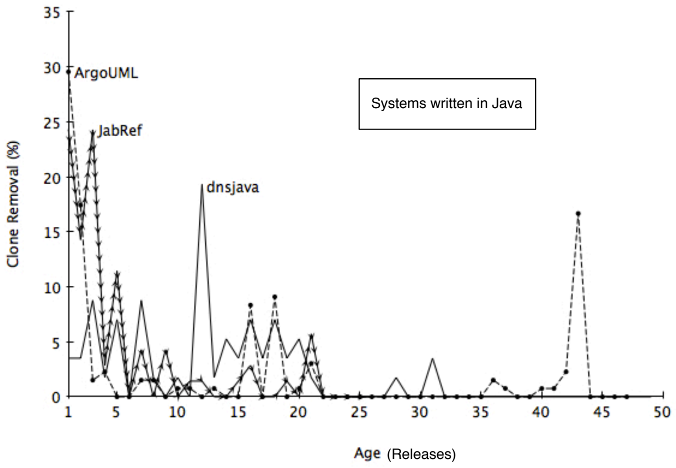
Ans. to RQ8: *During the evolution of the software systems, a few early releases experience significant clone removal. Nevertheless, some clones propagated over a relatively long sequence of releases before they were finally removed.*

This finding is also in keeping with the answer to the research question *RQ5* (Section 4.5), which indicates that most of the clones do not undergo frequent changes before their removal. We suspect that once a developer comes to know of a clone during its first change, this awareness might drive the removal of the clone at a later release. This indicates an area where informed clone management can play a significant role.

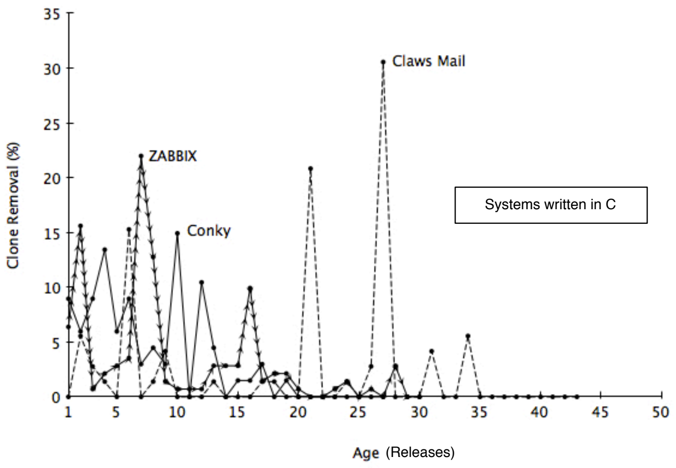
5. THREATS TO VALIDITY

In this section, we discuss possible threats to the validity of our study and how we mitigated their effects.

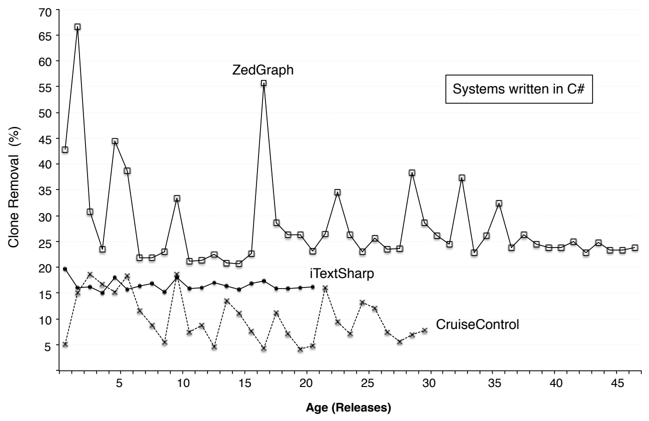
Construct Validity: Perhaps the best way to investigate change and evolution of clones is to study of the individual clone fragments in terms of genealogies across versions of the



(a)



(b)



(c)

Figure 3. Clone removal over sequences of releases

system. As versions one might choose programmers' commit transactions or weekly/monthly snapshots of the code base, or the stable releases of the system. A number of the earlier studies [8, 15] used the programmers' commit transactions or weekly/monthly snapshots of the code base, while many other studies [30, 31, 40] used software releases as the versions.

Programmers often create clones for experimental purposes, which they remove shortly after creation [15]. Thus, daily, weekly or monthly snapshots can be too frequent to capture stable changes in the code base. Indeed, commit transactions are more susceptible to this issue, in addition to their sensitivity to the developers' commit styles [40]. However, when a version of a software is officially released, the source code is expected to be in a stable form. Moreover, even a large number of weekly/monthly revisions may correspond to only a few stable releases, whereas series of releases typically span a longer period of development time. Therefore, for our study, we selected stable releases of the systems instead of commit transactions or snapshots at certain time intervals.

Internal Validity: The internal validity of our study is subject to the accuracy in clone detection and genealogy extraction. The `NiCad` clone detector used in our study, is reported to be effective in detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones with high precision and recall [28, 29]. Moreover, our manual verification of random samples from the detected clones found no false positives. The genealogy extractor `gCad`, used in our study, is also reported to be accurate in the computation of near-miss clone genealogies [31]. Nevertheless, we carried out manual investigation to verify the correctness of the genealogies and to fix any inconsistencies. Indeed, the manual assessment can be subject to human errors. However, all the human participants of this study are faculty and graduate students carrying out research in the area of software clones, and thus we believe that they have affluent expertise to keep the probable human errors to the minimum.

External Validity: Our study is based on nine medium to fairly large open-source software systems, and thus one may question the *generalizability* of the findings. However, for each of the subject systems, we studied a significant number of releases, and we expect this to help minimize the threat to some extent. To further mitigate the threat, we carefully chose the subject systems from different application domains, and written in different programming languages.

Reliability: The methodology of this study including the procedure for data collection are documented in this paper. The subject systems are open-source, while the `NiCad-2.6.3` clone detector and the `gCad` genealogy extractor are also available online[2]. Therefore, it should be possible to replicate the study.

6. RELATED WORK

There has been considerable research in characterizing clone evolution and distinguishing clones of interest for removal by refactoring.

[2]http://usask.ca/∼minhaz.zibran/pages/projects.html

From a manual analysis of 800 function/method level clones over six different open-source Java systems, Balazinska et al. [2] proposed a taxonomy of function clones, based on the differences and similarities in the program elements. On the basis of the location of clones in the inheritance hierarchy, Koni-N'Sapu [16] proposed another clone taxonomy and a set of object-oriented refactoring patterns for refactoring each category of code clones. Later, Kapser and Godfrey [13] proposed a clone taxonomy based on the locations of clones in the file-system hierarchy and (dis)similarities in the code functionalities.

Schulze et al. [33] proposed a code clone classification scheme to support the decision of whether to use Object-Oriented Refactoring (OOR) or Aspect Oriented Refactoring (AOR) for clone removal. Other techniques, such as design patterns [1] and traits [22] were also attempted to identify and refactor clones of interest. Torres [37] applied a concept-lattice based data mining approach to derive four categories of concepts containing duplicated code and suggested refactoring patterns suitable for refactoring clones in each of the categories.

Higo et al. [10] proposed a software-metrics-based approach to identify potential clones that can be easier to refactor using the *extract method* and *pull-up method* refactoring patterns. Variations of such metrics-based approaches are realized in tools namely `Gemini` [38] and `ARIES` [10]. Choi et al. [4] carried out a developer-centric study to determine the effectiveness of different combinations of metrics in distinguishing clones of interest for refactoring.

None of the aforementioned work was based on code clone genealogies as ours, where we examined the evolution of individual clone fragments to characterize the patterns of change and removal of clones. Based on the experience from an ethnographic study on copy and paste programming practices, Kim et al. [14] reported that "larger or frequently copied code fragments are good candidates for refactoring." The findings of our study also to support their conjecture.

Based on a case study on two open source Java systems, Tairas and Gray [35] reported that in some cases clone refactorings were partially performed on only parts of the clones (i.e., sub-clones). However, their focus was only on the occurrences of refactorings composed of the *extract method* refactoring pattern. The objective of our work was to investigate and characterize removal and refactoring of clones not only through the *extract method* refactoring patterns, but also by all other possible means.

Göde [8] conducted a case study over four systems, and investigated the extent clones were removed from the systems. He found many instances of deliberate clone removal, and the majority of those removals were performed by the *extract method* refactoring pattern. He further reported that the developers refactored mostly the closely located clones, which is also consistent with our findings.

The study of Göde was based on only three metrics, and he concluded that more complex metrics such as change frequency of clones should be examined to better understand the phenomenon. In our study, based on clone genealogies over 329 releases of nine software systems and using a wide range of characterization criteria, we captured a broader picture of clone removal and changes in open-source software systems.

7. CONCLUSION

This paper presents a genealogy-based empirical study on the evolution of individual clone fragments to characterize the changes and removal of exact (*Type-1*) and near-miss (*Type-2* and *Type3*) code clones. We examined a total of 329 releases from nine open-source software systems written in Java, C, and C#.

In the study, we addressed eight research questions, and derived answers to those with a combination of qualitative and quantitative analyses as well as statistical tests of significance. The findings of our study shed light on the conventional wisdom about clone evolution, in particular, derive useful insights into the patterns of changes and removals of code clones in practice.

From the study, we found that the sizes of the clone-groups (in terms of the number of member clone fragments), or the granularity (i.e., functions or blocks) of clones, or their dispersion in the file-system hierarchy do not have any significant effect on clone removal in practice. In terms of change patterns, we did not find any relationships between clone removal and any particular type of changes (i.e., consistent or inconsistent).

However, highly similar or larger clone fragments appear to be attractive for removal. A few early releases of the software systems experienced significantly more changes and removal of clones than the later releases. Inconsistent changes are found to have dominated over consistent changes of code clones. We also found that the majority of clones that were removed, did not experience frequent changes before removal, and surprisingly, most of those clones underwent changes only once, before they were removed from their respective systems.

During manual investigation, we discovered many instances of clones, which could be attractive for refactoring, but those were left alone, perhaps due to the lack of proper tool support. We believe that the practical findings from this study make significant contributions to the existing wisdom about clone evolution, refactoring, and removal, which in turn, will be useful for devising effective tools and techniques for informed clone management.

Acknowledgement: This work is supported in part by the Walter C. Sumner Memorial Foundation.

8. REFERENCES

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of the 7th Working Conference on Reverse Engineering*, pp. 98–107, 2000.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of the 6th International Symposium on Software Metrics*, pp. 292–303, 1999.

[3] D. Cai and M. Kim. An empirical study of long-lived code clones. In *Proc. of the International Conference on Fundamental Approaches to Software Engineering*, pp. 432–446, 2011.

[4] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proc. of the 5th International Workshop on Software Clones*, pp. 7–13, 2011.

[5] J. Cordy and C. Roy. The NiCad clone detector. In *Proc. of the Tool Demo Track of the 19th International Conference on Program Comprehension*, pp. 219–220, 2011.

[6] J. Cordy and C. Roy. Tuning research tools for scalability and performance: the NiCad experience. In *Science of Computer Programming*, 79(1):158–171, 2014.

[7] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of the 33rd International Conference on Software Engineering*, pp. 311–320, 2011.

[8] N. Göde. Clone removal: fact or fiction? In *Proc. of the 4th International Workshop on Software Clones*, pp. 33–40, 2010.

[9] N. Göde and J. Harder. Clone stability. In *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, pp. 65–74, 2011.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proc. of the 8th IASTED International Conference on Software Engineering and Applications*, pp. 222–229, 2004.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the 31st International Conference of Software Engineering*, pp. 485–495, 2009.

[12] C. Kapser and M. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software entities, In *Journal of Empirical Software Engineering*, 13(6):645–692, 2004.

[13] C. Kapser and M. Godfrey. Aiding comprehension of cloning through categorization. In *Proc. of the 7th International Workshop on Principles of Software Evolution*, pp. 85–94, 2004.

[14] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proc. of the International Symposium on Empirical Software Engineering*, pp. 83–92, 2004.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187–196, 2005.

[16] G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in OO systems. Diploma thesis, University of Bern, 116 pp., 2001.

[17] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the 8th International Conference on Source Code Analysis and Manipulation*, pp. 57–66, 2008.

[18] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proc. of the 4th International Workshop on Software Clones*, pp. 65–72, 2010.

[19] D. Anderson, D. Sweeney, and T. Williams. *Statistics for Business and Economics*. Thomson Higher Education,10th Edition, 2009.

[20] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proc. of the 27th ACM Symposium On Applied Computing (SE Track)*, pp., 1227–1234, 2012.

[21] M. Mondal, C. Roy, and K. Schneider. An empirical study on clone stability. In *Applied Computing Review*, 12(3):20–36, 2013.

[22] E. Murphy-Hill, P. Quitslund, and A. Black. Removing duplication from java.io: a case study using traits. In *Proc. of the ACM SIGPLAN conference on Systems, Programming, Languages and Applications*, pp. 282–291, 2005.

[23] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. In *IEEE Transaction on Software Engineering*, 38(5):1008–1026, 2011.

[24] J. Pate, R. Tairas, and N. Kraft. Clone evolution: a systematic review. In *Journal of Software: Evolution and Process* , 25(3): 261–283, 2013.

[25] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? In *Proc. of the 7th Working Conference on Mining Software Repository*, pp. 72–81, 2010.

[26] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proc. of the 11th Working Conference on Reverse Engineering*, pp. 100–109, 2004.

[27] C. Roy and J. Cordy. A survey on software clone detection research, *Technical Report 2007-541*, School of Computing, Queen's University, 115 pp., 2007.

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

[29] C. Roy and J. Cordy. NiCad: Accurate Detection of Near-Miss Intentional clones using flexible pretty-printing and code Normalization. In *Proc. of the 16th International Conference on Program Comprehension*, pp. 172–181, 2008.

[30] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: an empirical study. In *Proc. of the 10th International Conference on Source Code Analysis and Manipulation*, pp. 87–96, 2010.

[31] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proc. of the International Conference on Software Maintenace*, pp. 293–302, 2011.

[32] R. Saha, C. Roy, K. Schneider, and D. Perry. Understanding the evolution of Type-3 clones: an exploratory study. In *Proc. of the 10th Working Conference on Mining Software Repositories*, pp. 139–148, 2013.

[33] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *Proc. of the 1st Workshop on Refactoring Tools*, pp. 6:1–6:4, 2008.

[34] J. Svajlenko, C. Roy, and J. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proc. of the Tool Demonstration Track of the 7th International Workshop on Software Clones*, pp. 8–9, 2013.

[35] R. Tairas and J. Gray. Sub-clones: Considering the part rather than the whole. In *Proc. of the 9th International Conference on Software Engineering Research and Practice*, pp. 284–290, 2010.

[36] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. In *Journal of Empirical Software Engineering*, 15(1):1–34, 2009.

[37] R. Torres. Source code mining for code duplication refactorings with formal concept analysis. M.Sc. thesis, Vrije Universiteit Brussel, 53 pp., 2004.

[38] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. of the 9th International Symposium on Software Metrics*, pp. 67–76, 2002.

[39] R. Venkatasubramanyam, S. Gupta, and H. Singh. Prioritizing Code Clone detection results for clone management. In *Proc. of the 7th International Workshop on Software Clones*, pp. 30–36, 2013.

[40] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: an empirical study. In *Proc. of the 16th International Conference on Engineering of Complex Computer System*, pp. 295–304, 2011.

[41] M. Zibran and C. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proc. of the 11th International Conference on Source Code Analysis and Manipulation*, pp. 105–114, 2011.

[42] M. Zibran and C. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach. In *Proc. of the 19th International Conference on Program Comprehension*, pp. 266–269, 2011.

[43] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring. In *IET Software*, 7(3):167–186, 2013.

[44] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *Proc. of 5th the International Workshop on Software Clones*, pp. 75–76, 2011.

[45] M. Zibran and C. Roy. IDE-based real-time focused search for near-miss clones. In *Proc. of the 27th ACM Symposium On Applied Computing (SE Track)*, pp. 1235–1242, 2012.

[46] M. Zibran, R. Saha, C. Roy, and K. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proc. of the 28th ACM Symposium On Applied Computing (SE Track)*, pp. 1123–1130, 2013.

[47] M. Zibran and C. Roy. The road to software clone management: A survey. *Tech. Report 2012-03*, Department of Computer Science, University of Saskatchewan, Canada, pp. 1–62, 2012.

ABOUT THE AUTHORS:

Minhaz F. Zibran is a Ph.D. candidate at the Department of Computer Science, University of Saskatchewan, Canada. His research interests include various aspects of software engineering with particular focus on the detection, analysis, and management of code clones in evolving software systems. Minhaz has co-authored scholarly articles published in ACM and IEEE sponsored international conferences and reputed journals. Throughout his career, Minhaz also earned both teaching and industry experience. He has been actively involved in organizing international conferences (e.g., ICPC'2011, SCAM'2012, ICPC'2012, WCRE'2012, ICSM'2013) in his area of research. His scholarly excellence enabled him earning many scholarships and awards including the postgraduate scholarship from the Natural Science and Engineering Research Council (NSERC) of Canada.

Ripon K. Saha is a Ph.D. student in the Department of Electrical and Computer Engineering at The University of Texas at Austin. He received his B.Sc. degree in Computer Science and Engineering from Khulna University, Bangladesh and M.Sc. degree in computer science from University of Saskatchewan, Canada. His research interests include program analysis, mining software repositories, and empirical software engineering.

Chanchal Roy is an assistant professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution, including clone detection, analysis and management, reverse engineering, empirical software engineering, and mining software repositories. He served or has been serving in the program committee of major software engineering conferences (e.g., ICSM, WCRE, MSR, ICPC and SCAM). He served as the Finance Chair for ICPC’11, Tool Co-chairs for ICSM’12 and WCRE’12, Tool Chair for SCAM’12, Poster Co-chair for ICPC’12, Program Co-chair for IWSC’12, and Finance Chair for ICSM’13. He has been working as the General Chair for ICPC’14.

Dr. Kevin Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Research Lab at the University of Saskatchewan. Dr. Schneider has held appointments as Computer Science Department Head, Vice-Dean Science, and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology. Before joining the University in 2001, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. Dr. Schneider is a member of the ACM and IEEE CS, an elected member of the International Federation for Information Processing working group 2.7/13.4 on user interface engineering and past Prairie representative for the Canadian Association of Computer Science.