Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study

Minhaz F. Zibran[1] Ripon K. Saha[2] Chanchal K. Roy[3] Kevin A. Schneider[4]
minhaz.zibran@usask.ca[1], ripon@utexas.edu[2], croy@cs.usask.ca[3], kas@cs.usask.ca[4]
University of Saskatchewan, Canada[1 3 4] University of Texas at Austin, USA[2]

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 six 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% [30] of code clones, and the proportion may be as high as 50% [20]. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'13 March 18-22, 2013, Coimbra, Portugal.
Copyright 2013 ACM 978-1-4503-1656-9/13/03 ...$10.00.

Code cloning is a popular code reuse mechanism that is used to speedup the development process and facilitate independent evolution of similar program features. 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 [31]. Thus, code clones may have a significant impact on the development and maintenance of software systems.

Despite ongoing research on the positive [11, 27] and negative [19] effects of code clones, researchers and practitioners have come to an accord for the need of active and informed clone management [32, 33] 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 [11, 31], given the risks and efforts needed for such activities. In this regard, a number of classification schemes [2, 9, 12, 25], metric based selection approaches [1, 4, 8], and an effort model [31] 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 [6]. 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 [34].

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. Previous 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. [11] suggests that many clones are *volatile* (i.e., disappear shortly after they are created), while the study of Lozano and Wermelinger [14] 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 seven 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. [10] 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. [10].

RQ3: *For a group of clones, does the distribution of the clones in the file system hierarchy impact their removal in practice?* — Göde [6] 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: *During the evolution of the software systems, when does clone removal take place?* — This question addresses the aforementioned contradiction about the volatility of clones.

RQ6: *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 [7, 13, 16].

RQ7: *Does the granularity (entire function bodies or syntactic blocks) of clones make any difference in their removal in practice?* — A recent study of Göde [6] reported many instances of removal of block clones by *extract method* refactoring.

To address the research questions, we carry out a systematic study based on code clone *genealogy* [11, 24], which maps the individual clone fragments across their evolution over subsequent releases. We investigate the changes and removal of individual clones in 228 releases of six diverse open-source software systems written in Java 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.

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. To the best of our knowledge, such a genealogy based study including *Type-3* clones has not yet been performed.

The rest of this paper is organized as follows. In Section 2, we introduce the terminology and metrics used in our study and the rest of the paper. 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, 5, 6, 11].

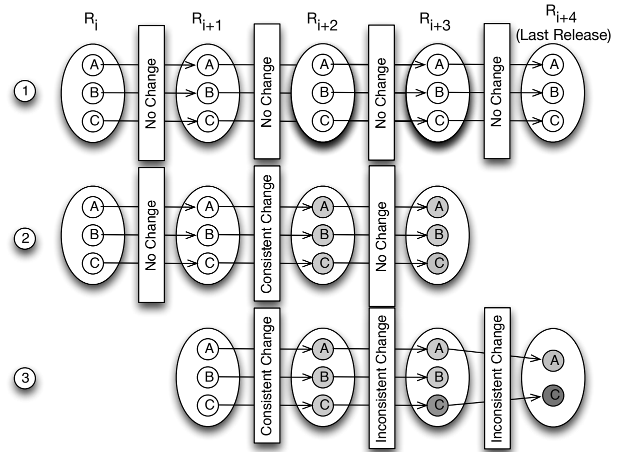


Figure 1: Different Types of Clone Genealogies

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*.

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.

Entropy of Dispersion: We used an entropy measure to characterize the physical distribution of clones at the file level by using Equation 1, where p_i denotes the probability of clones being located in file i.

$$entropy = \sum_{i=1}^{n} -p_i \log(p_i) \quad (1)$$

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 densed in only a few files. If the entropy is high, clones are scattered across different files.

Table 1: Software Systems Subject to Our Study

Prog. Lang.	System Name	Number of Releases	Releases Start	Releases End	Dates (mm/dd/yy) Start	Dates (mm/dd/yy) End	Duration (months)	LOC (range)
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

Table 2: `NiCad` Setting for Clone Detection

Setting	Value
Granularity of Clones	Block
Minimum Clone Size	5 LOC
Filtering of Statements	None
Renaming of Identifiers	Blind
UPI (dissimilarity) Threshold	30%

3. STUDY SETUP

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

3.1 Extraction of Genealogies

Using the `NiCad-2.6.3` [22] clone detector, we separately detected clones from every release of each of the subject systems. The parameter setting of `NiCad` that we used for clone detection, is presented in Table 2. With this setting, `NiCad` detects both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones.

From the clone detection results obtained from `NiCad`, for each of the subject systems, we separately constructed the clone genealogies using `gCad` [24]. The tool `gCad` is a clone genealogy extractor we developed, which can construct and classify genealogies of all three types (*Type-1, Type-2*, and *Type-3*) of clones that we are interested in. As per the need of this study, we significantly extended and customized the tool with a set of appropriate features to compute the necessary metrics.

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 [30, 31], 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 [15] with $\alpha = 0.05$, to determine the statistical significance of the findings.

Table 4: Average Sizes of Removed and Alive Clone-Groups

System	Removed Clones	Alive Clones
dnsjava	2.25	2.75
JabRef	2.31	4.17
ArgoUML	2.12	9.12
ZABBIX	2.31	4.53
Conky	2.37	9.31
Claws Mail	2.88	2.95

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 4). As seen in Table 4, 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 ($P = 0.008$) *rejects* (as, $P < \alpha$) the null hypothesis, which implies that the difference is 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 make a statistically significant difference in clone removal, and the smaller clone-groups are found to be attractive for removal in practice.*

4.2 Size of the Clone Fragments

The size 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 LOC per clone fragment in the clone-groups, we separately computed the average number of pretty-printed LOC 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 5.

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

Table 3: Actual and Normalized Textual Similarity of Removed and Alive Clone-Groups

Subject System	Actual Text Similarity				Normalized Text Similarity			
	Removed Clones		Alive Clones		Removed Clones		Alive Clones	
	Average	SD	Average	SD	Average	SD	Average	SD
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
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

SD = Standard Deviation

Table 5: Average Sizes (LOC) of Clone Fragments

Subject System	Removed		Alive	
	Average	SD	Average	SD
dnsjava	10	3	11	5
JabRef	17	15	13	9
ArgoUML	16	13	15	19
ZABBIX	26	25	21	21
Conky	20	23	15	7
ClawsMail	15	9	16	18

SD = Standard Deviation

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 ($P = 0.419$) over the series of sizes for the removed and alive clones *fails to reject* (as, $P > \alpha$) the null hypothesis.

As can be observed from Table 5, there are subtle differences in the sizes of the clone fragments of both the removed and alive clone-groups. For four of the subject systems (`ZabRef, ArgoUML, ZABBIX,` and `Conky`), the average sizes of clone fragments of removed clone-groups is higher than those of the alive clone-groups, which is just the opposite for the other two systems (`dnsjava` and `Claws Mail`). Hence, from the high level view the anticipation of Kim et al. [10] saying – developers are more interested in getting rid of larger clones – appears to be true. However, the high standard deviation in the average sizes of the removed clones indicates that developers remove varying sizes of clone fragments. Our manual investigation also confirms this finding. We found many instances where the developers removed clones consisting of less than seven LOC as well as clones with more than even 100 LOC. This makes it difficult to derive any relationship between clone removal and the sizes of the clone fragments. 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 not have a statistically significant impact on clone removal in practice, and the developers remove clone fragments of diverse sizes.*

4.3 Entropy of Dispersion

In Table 6, 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 colocated clones may required 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 6, 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.

Table 6: Comparison of Entropy of Dispersion

System	Removed Clones	Alive Clones
dnsjava	0.71	0.90
JabRef	0.53	0.98
ArgoUML	0.82	1.30
ZABBIX	0.35	0.53
Conky	0.18	0.24
Claws Mail	0.30	0.70

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 ($P = 0.199$) 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, 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

Table 7: Removal of Clone-Groups Classified by Change Patterns

Subject System	Static Clone-Groups			Consistently Changed CG			Inconsistently Changed CG		
	Total	Removed	[%]	Total	Removed	[%]	Total	Removed	[%]
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
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

Table 8: *MWW* Tests Over Removal of Categories of Clones

	Static	Consistently Changed	Inconsistently Changed
Static	-	$P = 0.378$	$P = 0.575$
Consistently Changed	$P = 0.378$	-	$P = 0.810$
Inconsistently Changed	$P = 0.575$	$P = 0.810$	-

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 7.

From Table 7, we see that for three of the systems (`JabRef`, `ZABBIX`, and `ClawsMail`), 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`. The results in the table do not give a clear indication whether a certain category of clone-group is removed more often. To obtain more confidence, 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 8, 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). This leads to the answer to the research question *RQ4* as follows.

Ans. to RQ4: *There is no statistically significant relationship between any particular type of changes in the clones, and their removal from a later release.*

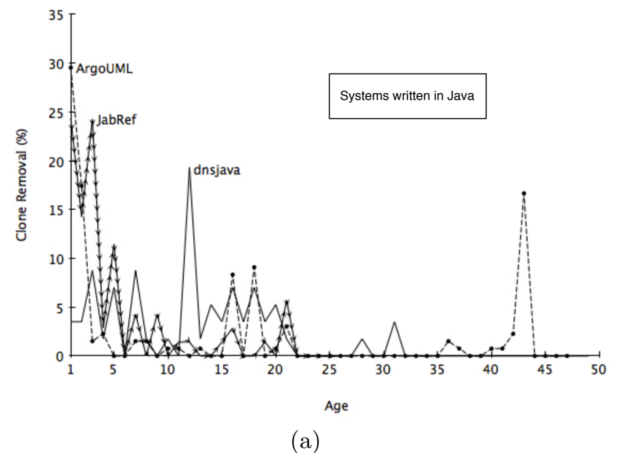
4.5 Age

The information about the age (in terms 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 subsequent release.

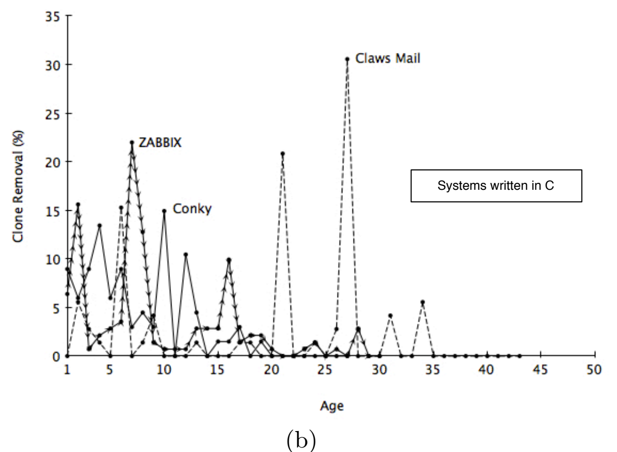
In Figure 2, we present the proportion of clone-groups found to have been removed in a subsequent release. As the figures (Figure 2(a) and Figure 2(b)) show, majority of the dead clones in four of the subject systems (`ArgoUML`, `JabRef`, `ZABBIX`, and `Conky`) were removed within the initial five to ten releases. This is consistent with that reported by Kim et al. [11], 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 *RQ5* as follows.

Ans. to RQ5: *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.*



(a)



(b)

Figure 2: Clone removal categorized by age

Table 9: Frequency of Changes before Removal

Subject System	Change Frequency 1	2	>2	Average
dnsjava	16	9	5	1.80
JabRef	11	4	3	1.72
ArgoUML	17	4	2	1.48
ZABBIX	30	16	7	1.74
Conky	10	8	5	1.95
ClawsMail	9	2	5	1.57

4.6 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 9, 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 *RQ6* as follows.

Ans. to RQ6: *Most clones do not undergo frequent changes before their removal.*

This finding is also in keeping with the answer to the research question *RQ5*, which indicates that many of the clones are removed from the systems within a few early releases. 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.

4.7 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

Table 10: Removal of Function and Block Clone Groups

Subject System	Function Clone Total	Rem.	[%]	Block Clone Total	Rem.	[%]
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
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

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 percentages of both the groups of function clones versus the block clones are presented in Table 10. 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 10 is lower than that of Table 7. Addressing the research question $RQ7$, we now formulate our null hypothesis as follows.

H_0^7**:** *The granularity (entire function bodies or syntactic blocks) of clones does NOT make any difference in their removal in practice.*

A MWW test ($P = 0.81$) over the proportions of the removal of both true function and block clones *fails to reject* (as, $P > \alpha$) the null hypothesis.

Table 10 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. Based on the above discussion, we now derive the answer to the research question *RQ7* as follows.

Ans. to RQ7: *In practice, the granularity (entire function bodies or syntactic blocks) of clones does not make any statistically significant difference in their removal.*

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 system. As versions one might choose programmers' commit

transactions or weekly/monthly snapshots of the code base, or the stable releases of the system. Programmers often create clones for experimental purposes, which they remove shortly after creation [11]. 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 [30]. However, when a version of a software is officially released, the source code is expected to be in a stable form. 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 [21, 22]. 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 [24]. 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 six 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 two 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` clone detector as well as the current extended version of `gCad` genealogy extractor are also available online[1]. 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 refactoring.

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. Based on the location of clones in the inheritance hierarchy, Koni-N'Sapu [12] proposed another clone taxonomy and a set of object-oriented refactoring patterns for refactoring each category of code clones. Later, Kapser and Godfrey [9] 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. [25] 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 [17] were also attempted to identify and refactor clones of interest. Torres [28] 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. [8] 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` [29] and `ARIES` [8]. 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. [10] reported that "larger or frequently copied code fragments are good candidates for refactoring." However, from our genealogy based study, we did not find evidence to support this conjecture.

Based on a case study on two open source Java systems, Tairas and Gray [26] 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 [6] 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. Our study significantly differs from that of Göde. Based on clone genealogies over 228 releases of six 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 228 releases of six open-source software systems written in C and Java. To the best of our knowledge, this is the first study in this regard that includes *Type-3* clone genealogies.

In the study, we addressed seven 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, on the patterns of changes and removals of code clones in practice.

From the study, we found that the sizes of the individual clone fragments or the clone-groups, or the granularity (i.e.,

[1] `http://www.cs.usask.ca/faculty/croy/`

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, a few early releases of the software systems experienced significantly more changes and removal of clones than the later releases. 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.

8. REFERENCES

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pp. 98–107, 2000.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pp. 292–303, 1999.

[3] D. Cai and M. Kim. An empirical study of long-lived code clones. In *FASE/ETAPS*, 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 *IWSC*, pp. 7–13, 2011.

[5] N. Göde and R. Koschke. Frequency and Risks of Changes to Clones. In *ICSE*, pp. 311–320, 2011.

[6] N. Göde. Clone Removal: Fact or Fiction? In *IWSC*, pp. 33–40, 2010.

[7] N. Göde and J. Harder. Clone stability. In *CSMR*, pp. 65–74, 2011.

[8] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *IASTED-SEA*, pp. 222–229, 2004.

[9] C. Kapser and M. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE*, pp. 85–94, 2004.

[10] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE*, pp. 83–92, 2004.

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

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

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

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

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

[16] 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 *ACM-SAC (SE Track)*, pp., 1227–1234, 2012.

[17] E. Murphy-Hill, P. Quitslund, and A. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA*, pp. 282–291, 2005.

[18] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Trans. on Softw. Engg.*, 1(1):1âĂŞ19, 2011.

[19] F. Rahman, C. Bird, P. Devanbu. Clones: What is that smell? In *MSR*, pp. 72–81, 2010.

[20] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-wide Code Duplication. In *WCRE*, pp. 100–109, 2004.

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

[22] C. Roy and J. Cordy. NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, pp. 172–181, 2008.

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

[24] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *ICSM*, pp. 293–302, 2011.

[25] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *WRT*, pp. 6:1–6:4, 2008.

[26] R. Tairas and J. Gray. Sub-clones: Considering the Part Rather than the Whole. In *SERP*, pp. 284–290, 2010.

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

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

[29] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *METRICS*, pp. 67–76, 2002.

[30] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study. In *ICECCS*, pp. 295–304, 2011.

[31] M. Zibran and C. Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *SCAM*, pp. 105–114, 2011.

[32] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *IWSC*, pp. 75–76, 2011.

[33] M. Zibran and C. Roy. IDE-based real-time focused search for near-miss clones. In *ACM-SAC (SE Track)*, pp. 1235–1242, 2012.

[34] 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.