An Empirical Study of the Fault-Proneness of Clone Mutation and Clone Migration

Shuai Xie[1], Foutse Khomh[2], Ying Zou[1]
[1] Department of Electrical and Computer Engineering, Queen's University, Canada.
{shuai.xie, ying.zou}@queensu.ca
[2] SWAT, École Polytechnique de Montréal, QC, Canada.
foutse.khomh@polymtl.ca

***Abstract*—When implementing new features into a software system, developers may duplicate several lines of code to reuse some existing code segments. This action creates code clones in the software system. The literature has documented different types of code clone (*e.g.*, Type-1, Type-2, and Type-3). Once created, code clones evolve as they are modified during both the development and maintenance phases of the software system. The evolution of code clones across the revisions of a software system is known as a clone genealogy. Existing work has investigated the fault-proneness of Type-1 and Type-2 clone genealogies. In this study, we investigate clone genealogies containing Type-3 clones. We analyze three long-lived software systems APACHE-ANT, ARGOUML, and JBOSS, which are all written in JAVA. Using the NICAD clone detection tool, we build clone genealogies and examine two evolutionary phenomena on clones: the *mutation* of the type of a clone during the evolution of a system, and the *migration* of clone segments across the repositories of a software system. Results show that 1) *mutation* and *migration* occur frequently in software systems; 2) the mutation of a clone group to Type-2 or Type-3 clones increases the risk for faults; 3) increasing the distance between code segments in a clone group also increases the risk for faults.**

***Index Terms*—Types of clones; clone genealogy; clone migration; fault-proneness.**

I. INTRODUCTION

Two or more code segments are considered to be clones when they have a high similarity or they are exactly the same. A clone pair consists of two code segments that are clones. A clone group is a set of cloned code segments, where any two of them can form a clone pair. A clone group is also called clone class. Two kinds of similarities are considered when defining code clones: textual similarity and functional similarity. Two code segments are considered similar when their program texts are similar or when their functionalities are similar. Based on the textual and functional similarity of code segments, clones have been classified in four types [1]:

- Type-1: Identical code segments except for variations in whitespace, layout and comments.
- Type-2: Syntactically identical segments except for variations in identifiers, literals, types, whitespace, layout and comments.
- Type-3: Copied segments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
- Type-4: Code segments that perform the same computation but that are implemented through different syntactic variants.

Developers often introduce clones either through copy and paste actions, or inadvertently. This code duplication is done within or between files contained in the same directory or across multiple directories. As developers duplicate code segments across parts of the system, many cloning styles emerge. For example, a developer may decide to restrict code duplication within a directory, while another may duplicate code across multiples directories. After a code duplication, the resulting clones are often modified during subsequent revisions of the system. A change to a clone group is considered consistent when the similarity between the code segments is preserved. Otherwise the change is inconsistent. Therefore, a consistent change on a clone group preserves the clone relationship between the code segments, but can change the type of the clone group (*e.g.*, a Type-1 clone group can become a Type-2 clone group as the result of a consistent change). We refer to this phenomenon as clone *mutation*.

The evolution of code clones across the revisions of a software system is known as a clone genealogy. This genealogy captures the creation, the propagation, and the evolution of code clones by developers [2]. The location in the source code directory structure of a clone segment can change during software evolution. We refer to this phenomenon as clone *migration*.

Many studies have investigated clone genealogies [3], [4]. However, very few have analyzed the impact of clone *mutation* and clone *migration* on the fault-proneness of software systems. In our study, we investigate the fault-proneness of genealogies containing *mutated* or *migrated* clones. We want to identify risky *mutations* and *migrations* of clones to warn development teams and guide their review and testing efforts.

In this work, we analyze clone genealogies containing Type-1, Type-2, and Type-3 clones, extracted from three large open source software systems written in JAVA, *i.e.*, ARGOUML, APACHE-ANT, and JBOSS. We do not work on Type-4 clones because to the best of our knowledge, no existing tool can successfully detect Type-4 clones [5]. We address the following three research questions:

RQ1: Do clone mutation and clone migration occur frequently in software systems?

We observe that clone *mutation* and clone *migration* affect respectively 31% and 48% of clone genealogies in JBOSS, 61% and 56% of clone genealogies in APACHE-ANT, and 40% and 68% of clone genealogies in ARGOUML. Overall, the two evolution phenomena affect an important number of clones and are therefore worth investigating further.

RQ2: Are some clone mutations more fault-prone than others?

We analyze whether clone groups migrated to certain types of clones are more fault-prone than others. Results show that clone genealogies predominated by Type-2 or Type-3 clones are more prone to faults than clone genealogies predominated by Type-1 clones. The mutation of Type-1 clones to Type-2 or Type-3 clones increases the risk for faults. However, when all the clone types in a clone genealogy are equally frequent, the risk for faults is reduced.

RQ3: Are some clone migrations more fault-prone than others?

We use the metric proposed by Kamiya *et al.* [6] to measure the distance between every two code segments (contained in a clone group) and identify different *migration* patterns followed by clone groups. We analyze whether clone groups following certain *migration* patterns are more fault-prone than others. Results show that in general, clone groups involved in migration patterns characterized by an increase of the distance between cloned code segments, are more fault-prone than others. Globally, a modification of the location of cloned code segments during the revisions of a software system increases the risk for faults in the system.

The rest of this paper is organized as follows. Section II discusses the related literatures on clone genealogies. Section III discusses the phenomena of mutation and migration, as well as our patterns of clone genealogies. Section IV explains the design of our study. Section V analyzes the results of our study. Section VI discusses threats to the validity of our study. Finally, Section VII concludes the paper and outlines some avenues for future research.

II. RELATED WORK

The first study on clone genealogies is proposed by Kim *et al.* [2]. They analyze the clone groups, known as clone classes, and define patterns of clone evolution. Using two JAVA systems and the CCFINDER clone detection tool, they perform a case study of the evolution of clones in software systems and conclude that at least half of clones in a software system are eliminated within eight check-ins after their creation. However, they do not consider the types of the clones in their study.

Our work strives for a deeper understanding of the evolution of different types of clones. We aim to identify risky clone evolution patterns in order to help developers better focus their maintenance efforts.

Barbour *et al.* [7] investigate late propagation genealogies in two open source software systems. They identify eight types of late propagation, which they classify in three groups based on the propagation of changes among the cloned code segments. They also analyze the fault-proneness of the eight types of late propagation and conclude that they are not equally risky. Only some late propagation genealogies require monitoring; in particular the late propagations involving no propagation are riskier than others. A late propagation clone pair is considered as involving no propagation when the clone pair is diverged and then reconciled without changes to the other clone segment in the pair.

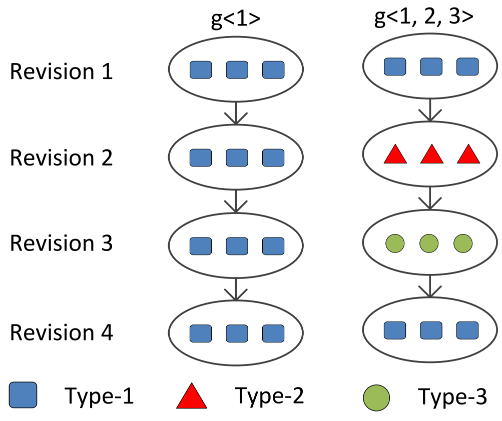


Figure 1. Example of Clone Genealogy

Aversano *et al.* [8], examine clone genealogies in two software systems to understand how clones are maintained. They select a specific stable snapshot of each of their studied systems and trace the evolution of cloned codes over time. They observe that about 18% of the clones exhibit a late propagation behavior. Out of the 17 instances of bug fixes in their data set, 7 occur during a late propagation. These founding suggest that late propagation genealogies are risky. In the same line, Thummalapenta *et al.* [3] perform a study on four open source C and JAVA systems and found that late propagation genealogies occur in a maximum of 16% of clone genealogies. They observe that clones with a late propagation genealogy are more prone to faults than others.

These studies are limited to Type-1 and Type-2 clones. Our work investigates clone genealogies in general, including genealogies containing Type-3 clones.

Zibran *et al.* [9], analyze the evolution of near-miss clones (*i.e.*, Type-2 and Type-3 clones) in software systems at release level using the NICAD clone detection tool. They conclude that the number of clones in a software system increases with the number of methods, but there is no relationship between clone density and the number of method. Our work also use the NICAD clone detection tool to identify clones but at revision level. We examine the likelihood of faults in clone genealogies containing Type-1, Type-2, and Type-3 clones at revision level.

III. CLONE GENEALOGIES

Code clones are often modified during subsequent revisions of a system. During these revisions, the type of a clone can change, we refer to this phenomenon as *clone mutation*. For example, a group of Type-1 clones can become Type-2 clones. Figure 1 presents the example of two clone genealogies.

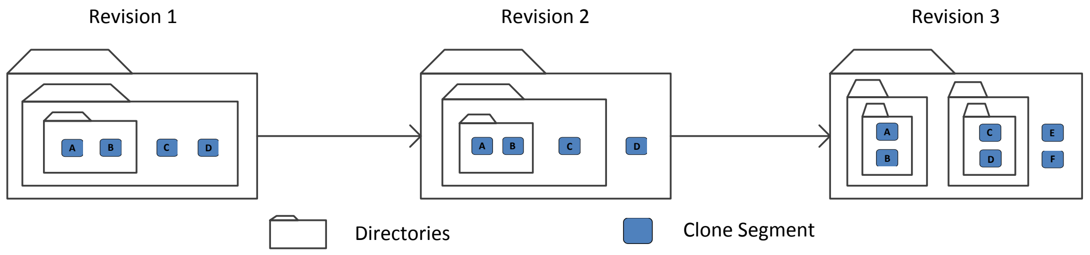


Figure 2. Example of Clone Migration

Table I
Categories of Clone Genealogies

Categories	List of clone types in the genealogy
G<1>	Type-1
G<2>	Type-2
G<3>	Type-3
G<1, 2>	Type-1, Type-2
G<1, 3>	Type-1, Type-3
G<2, 3>	Type-2, Type-3
G<1, 2, 3>	Type-1, Type-2, Type-3

For the clone genealogies G<1>, the type of the clone is remained unchanged during four consecutive revisions, while for the genealogy G<1, 2, 3>, the type of the clones is transitioned from Type-1 to Type-2 at revision 2, from Type-2 to Type-3 at revision 3, and then back to Type-1 at revision 4. Considering *clone mutation*, we organize clone genealogies in seven categories as presented in Table I. Each category of clone genealogy is characterized by the list of clone types involved in the genealogy. For example, G<1, 2> represents the category of clone genealogies where clones are transitioned from Type-1 to Type-2 or reciprocally, during the evolution of the system.

The location of a clone can also change during the revisions of the system. We refer to this phenomenon as *clone migration*. Figure 2 presents an example of clone migration. In this example, the clone segment D is moved to a parent directory at revision 2. D is still a clone of A, B, and C, but is now contained into a file located in a different directory. At revisions 3, the clone segments D and C are moved to newly created directories and two new code segments E and F, similar to A, B, C, and D are added to the clone group. To capture the migration of clones across the directories of a software system, we compute the distance between every two clone segments in the code directory structure as defined by Kamiya *et al.* [6].

For two clone segments A and B belonging to the clone group G, if f_1 and f_2 are the files containing A and B. The distance between A and B, $d_{dir}(A, B)$ is $d_{dir}(f_1, f_2)$. So $d_{dir}(A, B)$ is the value of the highest (first) level in the smaller (shorter) one of two paths have a different folder. The value of the level is counted in the smaller path, where the first folder in the directory is considered as highest level. The lowest level for a path of a file is 1 (*e.g.*, If A and B are included in the same file, $d_{dir}(A, B) = 0$).

Table II
Clone Migration Patterns.

Pattern	Description	
	Evolution trend of the	
	median distance of the clone group	size of the clone group
Constant	Constant	Constant
Wave Stable	Constant	Increase, Decrease, Wave Increase, or Wave Decrease
	Wave Constant	Constant, Increase, Decrease, Wave Increase, or Wave Decrease
High Density Strong Up	Increase	Increase or Wave Increase
Low Density Strong Up	Increase	Constant, Decrease, or Wave Decrease
High Density Wave Up	Wave Increase	Increase or Wave Increase
Low Density Wave Up	Wave Increase	Constant, Decrease, or Wave Decrease
High Density Strong Down	Decrease	Decrease or Wave Decrease
Low Density Strong Down	Decrease	Constant, Increase or Wave Increase
High Density Wave Down	Wave Decrease	Decrease or Wave Decrease
Low Density Wave Down	Wave Decrease	Constant, Increase or Wave Increase

Based on the variation of the size of a clone group and the variation of the median distance between the code segments in the clone group, during the evolution of the software system, we have identified 10 *migration* patterns described in Table II.

Each row in Table II presents the name of a migration pattern and a description of the pattern in terms of variations observed respectively on the median distance among all code segments in a clone group and on the size of the clone group. For example, the fourth row of Table II presents the *High Density Strong Up* migration pattern which is observed when the median distance among all the code segments in a clone group and the number of code segments (*i.e.*, the size) in a clone group increases continuously during the evolution of a software system.

At Row 5, the median distance between the code segments of a clone group experiencing the *Low Density Strong Up* migration pattern also increases continuously, while the size of the clone group either decreases or remains constant. The term “Wave” is used to describe a sequence of increase and decrease. A “Wave Increase” (respectively Decrease) is a sequence of increase and decrease where the final value is greater (respectively lower) than the initial value.

IV. Study Design

This section presents the design of our case study, which aims to address the aforementioned three research questions.

The *goal* of this study is to assess the risk for faults when *mutating* and *migrating* Type-1, Type-2, and Type-3 clones. The *quality focus* is the increase in maintenance effort and cost due to the presence of these clones in software systems. The *perspective* is that of researchers, interested in studying the risk of faults caused by the *mutation* and the *migration* of Type-1, Type-2, and Type-3 clones, during the evolution of software systems. The results of this study can also be of interest to developers, who perform development or maintenance activities and need to take into account and forecast their effort, and to testers, who need to know which code segments are important to test.

A. Data Collection

In this study, we analyze the change history of three software systems, ArgoUML, Apache-Ant, and JBoss which have different sizes, are written in same programming languages but belong to different domains. Table III presents some descriptive statistics of the systems.

Table III
Characteristics of the Systems

System	# LOC	# Revisions	# Group Genealogies
JBoss	1.6M	109K	1.7K
Ant	2.3M	1.0M	23 K
ArgoUML	3.1M	18K	15.6K

JBoss is a Java-based open source application server. It was created in 1999 and is still developed as a division of Red Hat. JBoss has 1.7M LOC and about 109K revisions in its software repository. We study code snapshots in the period from April 2000 to December 2010.

Apache-Ant is a Java library and a tool to compile, assemble, test and run Java, C and C++ applications. The project started in January 2000 and is still active. It is written in Java and has over 2.3M LOC and 1.0M revisions in its revision history. We study code snapshots from the period of January 2000 to November 2010.

ArgoUML is a UML-modelling application for forward and reverse engineering of source code. It provides a user with a set of views and tools to model systems using UML diagrams, to generate the corresponding code skeletons, and to reverse-engineer diagrams from existing code. The project started in January 1998 and is still active. It is written in Java and has over 3.1M LOC and 18K revisions in its software repository. We study code snapshots from the period of January 1998 to November 2010. ArgoUML has been used in previous studies on code clone evolution [7], [8].

B. Data Processing

Figure 3 shows an overview of our data processing approach. We follow the same steps as in our previous work [4] to build clone genealogies. More specifically, we use the tool J-Rex [10] to mine the source code repository of each Java subject system. The tool J-Rex identifies the revisions that modify each Java file and outputs a snapshot of the files at those revisions. Revisions corresponding to fault fixes are marked during the process. Next, we remove test files and perform clone detections on the systems using the NiCad clone detection tool. We map the obtained clones across the revisions to create clone genealogies. Finally, we categorize clone genealogies based on clone types as described in Section III. The remainder of this section elaborates more on these steps.

1) Mining Version Control Systems to Identify Faults: We use J-Rex to extract snapshots of our subject systems at each revisions. For each snapshot, we flag all the methods that have been modified since the last revision of the system. Again using J-Rex, we analyze each commit message to identify fault fixing commits. J-Rex implements the heuristics proposed by Mockus *et al.* [11] to identify fault fixing changes. The accuracy of J-Rex was reported to be 87% [7]. An empirical study conducted with professional developers by Hassan [12] reported that the ability of J-Rex to recognize fault fixes is comparable to that of a professional developer. The correlation between J-Rex and these professional developers was found to be > 0.8.

2) Removing Test Files: For the same reasons as Barbour *et al.* [7], we remove test files from our study. In fact, test files are only used to test the functionalities of the system. They are often copied and changed to test different execution scenarios, and consequently contain many clones that are not involved in normal executions of the system.

3) Detecting Clones: After removing the test files, we process the snapshots of all remaining files to extract the methods. We save each method snapshot in an individual file and submit to the NiCad clone detection tool. Similar to Göde *et al.* [13] and Barbour *et al.* [7], we exclude package and import statements, which may produce clones that are not interesting for developers. We also enforce hard boundaries between methods to avoid clones beginning in one method and ending in another; such clones are syntactically incorrect.

NiCad [14] is a flexible TXL-based hybrid language-sensitive and text comparison software clone detection tool capable of handling C, C-Sharp, Java, Python and WSDL languages. Roy *et al.* [14] report that NiCad can detect both exact (*i.e.*, Type-1) and near-miss (*i.e.*, Type-2 and Type-3) clones with high precision and recall. We select NiCad for our study because it is fast and consume very little memory. We need to parse all the revisions in one shot. Abstract syntax-based tools, such as CloneDR [1], require extensive memory and computation powers. Hence, they have limitations when scanning the entire history of a system.

[1]http://www.semdesigns.com/Products/Clone/

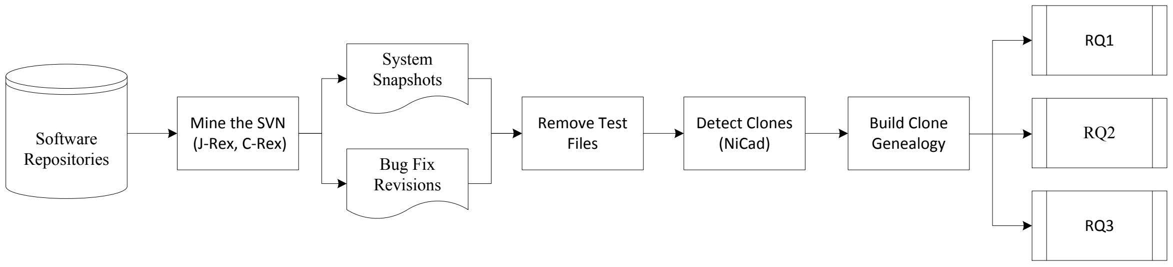


Figure 3. Overview of the Analysis Process

Table IV
NICAD'S PARAMETERS

Clone Types	Identifier Renaming	Similarity Threshold
Type-1	None	100%
Type-2	Blind-rename	100%
Type-3	Blind-rename	80%

NICAD has been used in a previous study on clone genealogy (*i.e.*, [9]). To allow our results to be comparable to those of previous studies, we configure NICAD using the same parameters as Zibran *et al.* [9]. Table IV shows the parameters of NICAD for this study. We use the 80% as the similarity to identify Type-3 clones, but we also check the results for other similarities. We use the latest version 3.4 of NICAD. We post-process the results of the clone detection to identify any clones that co-exist within the same revision. This approach is similar to [7], [15].

4) Building Clone Genealogies: Before building clone genealogies, we remove all unchanged clones. We also remove clones where all code segments belong to the same method, since these are invalid clones. After this cleaning step, we map all the clones reported by NICAD across the revisions of the system following our approach described in [7]. This approach consists the following steps: First, we query the output of J-REX to obtain the list of all the revisions where the methods containing the clones were modified. For each modified method, we identify if the contained clones were changed. We repeat the entire process for each revision in the revisions list, until all the revisions are visited.

C. Statistical Analysis Method

We use the Chi-Square test [16] to determine if there are non-random associations between a particular type of clone genealogy and the occurrence of future bugs. We use the 5% level (*i.e.*, p-value <0.05) to identify significant results of the Chi-square test. We also compute *odds ratio* (OR) [16] which indicates the likelihood of an event to occur (*i.e.*, a fault fixing change) in one sample (*i.e.*, experimental group), to the odds q of the event to occur in the other sample (*i.e.*, control group): $\text{OR} = \frac{p/(1-p)}{q/(1-q)}$. An $OR = 1$ indicates that the event is equally likely in both samples; an $OR > 1$ shows that the event is more likely in the experimental group while an $OR < 1$ indicates the opposite, *i.e.*, the event is more likely in the control group.

V. CASE STUDY RESULTS

This section presents and discusses the results of our research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

RQ1: Do clone mutation and clone migration occur frequently in software systems?

Motivation. This question is preliminary to **RQ2** and **RQ3**. It aims at providing quantitative evidences of the occurrence of clone *mutation* and clone *migration* in our studied systems. If these two phenomenons are very frequent, then they are worth studying in more details in order to advise developers and managers about potential side effects resulting from them.
Approach. We address this question by extracting clone genealogies from our subject systems following the method described in Section IV-B4, and classifying the genealogies using the categories described in Table I (for *mutation*) and Table II (for *migration*). For each category of clone genealogy, we report the number of occurrences in the systems.

Table V
NUMBER OF CLONE GROUPS THAT UNDERWENT A MUTATION

Categories	JBoss		Apache-Ant		ArgoUML	
	number	%	number	%	number	%
G<1>	71	4.27	2565	11.17	251	1.61
G<2>	587	35.34	1994	8.68	5706	36.54
G<3>	492	29.62	4497	19.59	3348	21.44
G<1,2>	195	11.74	7189	31.31	5193	33.25
G<1, 3>	184	11.08	3632	15.82	471	3.02
G<2, 3>	120	7.22	1999	8.71	459	2.94
G<1, 2, 3>	12	0.72	1085	4.73	188	1.20
Total	**1661**	**100**	**22961**	**100**	**15616**	**100**

Findings. Table V lists the seven categories of clone genealogies identified in Section III and the proportion of their occurrences for each of our subject systems. For each system and for each category, we presents the number and the percentage of genealogies in that category. Table VI presents for each of the systems, the proportion of clone genealogies that followed one of the ten migration patterns identified in Section III.

As summarized in Table V, 30.77% of clone genealogies in JBOSS experienced a clone mutation (*i.e.*, G<1, 2>, G<1,

Table VI
NUMBER OF CLONE GENEALOGIES THAT FOLLOWED ONE OF THE TEN MIGRATION PATTERNS

Migration patterns	JBoss		Apache-Ant		ArgoUML	
	number	%	number	%	number	%
Constant	865	52.08	10107	44.02	4921	31.51
Wave Stable	317	19.08	5401	23.52	5543	35.50
High Density Strong Up	44	2.65	220	0.96	28	0.18
Low Density Strong Up	182	10.96	66	0.29	57	0.37
High Density Wave Up	40	2.41	1751	7.63	2118	13.56
Low Density Wave Up	173	10.42	506	2.20	216	1.38
High Density Strong Down	4	0.24	1682	7.33	558	3.57
Low Density Strong Down	23	1.38	515	2.24	48	0.31
High Density Wave Down	0	0.00	1617	7.04	298	1.91
Low Density Wave Down	13	0.78	1096	4.77	1829	11.71
Total	**1661**	**100**	**22961**	**100**	**15616**	**100**

3>, G<2, 3>, or G<1, 2, 3>). In APACHE-ANT, 60.56% of clone genealogies are concerned by a clone mutation, and in ARGOUML, 40.41% of clone genealogies are concerned by a mutation. The most frequent form of clone *mutation* is between Type-1 and Type-2 clones (*i.e.*, G<1, 2>).

For clone *migration*, during the evolution of JBOSS, the directory of at least one clone segment was changed during the evolution of 48% of the clone groups. In APACHE-ANT and ARGOUML, respectively 56% and 68% of clone groups experienced a *migration*. The most frequent migration pattern is the Wave Stable pattern.

Overall, we conclude that *mutation* and *migration* are two phenomena that affect an important number of clones in software systems.

In the next two research questions, we examine these two phenomena in more detail to determine if some types of *mutation* and *migration* are more risky than others.

RQ2: Are some clone mutations more fault-prone than others?

Motivation. Development teams are interested in identifying areas in their software systems that are more likely to contain faults. Cloned code have been reported to contain more faults than non-cloned code [3], [7], [8]. Because a software system can contain up to 20% of cloned code [5], it can be very expensive to monitor all cloned code in a software system. Therefore, it will be interesting for development teams to identify clones that are most at risk of faults, in order to allocate their limited testing and review resources towards these clones. A change that modifies the type of a clone group (*i.e.*, a *mutation*) could affect the ability of developers' to keep track of all related clone segments in the clone group. Developers may also have trouble propagating changes to all clone segments in the group consistently; resulting in an increased risk for faults. In this research question we examine the *mutation* of clones during software evolutions. More precisely, we analyze the fault-proneness of the seven categories of clone genealogies identified in Section III. We aim to identify risky types of clone *mutations* that should be highlighted for monitoring.

Approach. For each system, we build clone genealogies following our method described in Section IV-B4. Next, we classify clone genealogies based on the types of the clones involved, as in Table I. For each system and each category, we compute the number of fault-containing and fault-free genealogies. We use the Chi-square test and compute Odds ratio to test the following null hypothesis[2]: H_{02}*: Each category of clone genealogy has the same proportion of clones that experience a fault fix*. When computing Odds ratios, we select the category of clone genealogies containing only Type-1 clones (*i.e.*, G<1>), as the control group. We form one experimental group for each of the remaining categories. We perform the Chi-square test using the 5% level (*i.e.*, p-value < 0.05).

Because our detection of Type-3 clones is done with a selected similarity threshold of 80% (see Table IV). We perform a sensitivity analysis to assess the impact of this chosen threshold on the results. Precisely, we repeat the detection of Type-3 clones using similarity threshold values of respectively 95%, 90%, 85%, 75% and 70%. For each of these similarity thresholds, we build clone genealogies, classify them following the categorization described in Table I, and repeat the testing of H_{02} using the Chi-square test and Odds ratios.

Findings. Table VII shows the results of the Chi-square test and lists *ORs* for the seven categories of clone genealogies described in Table I. For each system, the control group is the category of clone genealogies containing Type-1 clones only. The Chi-square test is statistically significant for JBOSS, APACHE-ANT and ARGOUML. Figure 4 presents the results of the sensitivity analysis. All the results presented on Figure 4 are statistically significant. Overall, we can reject H_{02}.

Genealogies G<1>, G<2>, or G<3>: For JBOSS and ARGOUML, results show that clone genealogies containing only Type-2 clones (*i.e.*, G<2>) are more prone to faults than clone genealogies containing either Type-1 or Type-3 clones only. For APACHE-ANT, clone genealogies containing only Type-3 clones (*i.e.*, G<3>) are more fault-prone. All these results are confirmed by the sensitivity analysis, with the exception of ARGOUML, where genealogies G<3> containing Type-3 clones, detected with the 70% similarity threshold are more fault-prone than G<2> genealogies

Genealogies G<1, 2>: ORs for genealogies containing Type-

[2]There is no H_{01} because **RQ1** is exploratory

Table VII
CONTINGENCY TABLE AND CHI SQUARE TEST RESULTS FOR THE CATEGORIES OF CLONE GENEALOGIES

Genealogies		JBoss			Apache-Ant			ArgoUML		
Categories	Most Frequent Clone Type	# of Genealogies with		ORs	# of Genealogies with		ORs	# of Genealogies with		ORs
		Faults	No Faults		Faults	No Faults		Faults	No Faults	
G<1>	Type-1	21	50	1	364	2201	1	115	136	1
G<2>	Type-2	293	294	2.37	745	1249	3.61	3355	2351	1.69
G<3>	Type-3	224	268	1.99	2159	2338	5.58	1550	1798	1.02
G<1, 2>	Type-1	3	0	-	186	723	1.56	34	23	1.75
	Type-2	50	3	39.68	1091	1091	6.05	1649	2157	0.90
	Type-1, Type-2	95	44	5.14	890	3208	1.68	573	757	0.90
G<1, 3>	Type-1	6	0	-	28	86	1.97	9	10	1.06
	Type-3	73	14	12.41	1447	1078	8.12	177	135	1.55
	Type-1, Type-3	41	50	1.95	261	732	2.16	64	76	1
G<2, 3>	Type-2	2	1	4.76	55	120	2.77	38	19	2.37
	Type-3	25	29	2.05	480	648	4.48	117	89	1.55
	Type-2, Type-3	24	39	1.46	267	429	3.76	93	103	1.07
G<1, 2, 3>	Type-1	0	0	-	15	30	3.02	4	0	-
	Type-2	0	0	-	24	50	2.90	21	37	0.67
	Type-3	8	0	-	283	454	3.77	39	36	1.28
	Type-1, Type-2, Type-3	4	0	-	70	159	2.66	25	26	1.14
p-values		**<0.05**			**<0.05**			**<0.05**		

Table VIII
RESULTS OF THE CHI-SQUARE TEST FOR THE TEN MIGRATION PATTERNS

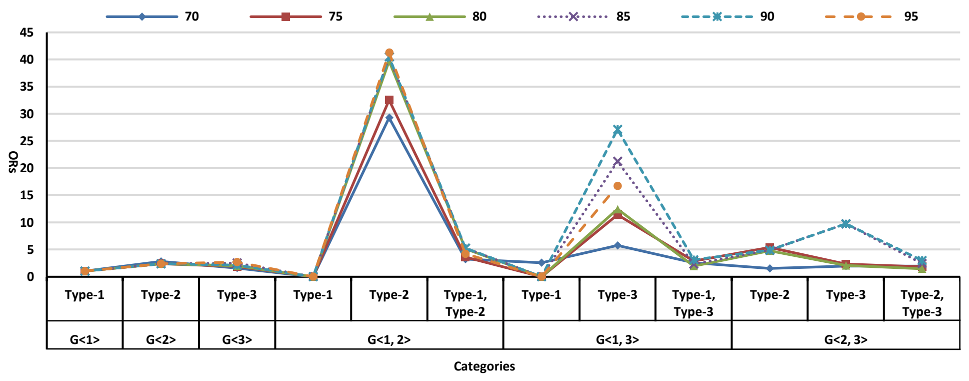
Migration patterns	JBoss			Apache-Ant			ArgoUML		
	# Faults	# No Faults	ORs	# Faults	# No Faults	ORs	# Faults	# No Faults	ORs
Constant	370	495	1.00	3709	6398	1.00	2986	1935	1.00
Wave Stable	156	161	1.30	1991	3410	1.01	2939	2604	0.73
High Density Strong Up	30	14	2.87	95	125	1.31	16	12	0.86
Low Density Strong Up	110	72	2.04	25	41	1.05	38	19	1.30
High Density Wave Up	39	1	52.18	755	996	1.31	876	1242	0.46
Low Density Wave Up	152	21	9.68	192	314	1.05	111	105	0.69
High Density Strong Down	1	3	0.45	301	1381	0.38	209	349	0.39
Low Density Strong Down	3	20	0.20	146	369	0.68	18	30	0.39
High Density Wave Down	0	0	-	506	1111	0.79	226	72	2.03
Low Density Wave Down	8	5	2.14	645	451	2.47	444	1385	0.21
p-values	**<0.05**			**<0.05**			**<0.05**		

1 and Type-2 clones are in general higher than ORs for genealogies containing either Type-1 or Type-2 clones only; meaning that the mutation of a clone from Type-1 to Type-2 or conversely increases the risk for faults. For JBOSS and APACHE-ANT, G<1, 2> genealogies predominated by Type-2 clones are more risky. For ARGOUML, G<1, 2> genealogies predominated by Type-1 clones are more risky. When Type-1 and Type-2 clones are equally frequent in a G<1, 2> genealogy, this risk for fault is reduced. This finding is confirmed by the sensitivity analysis.

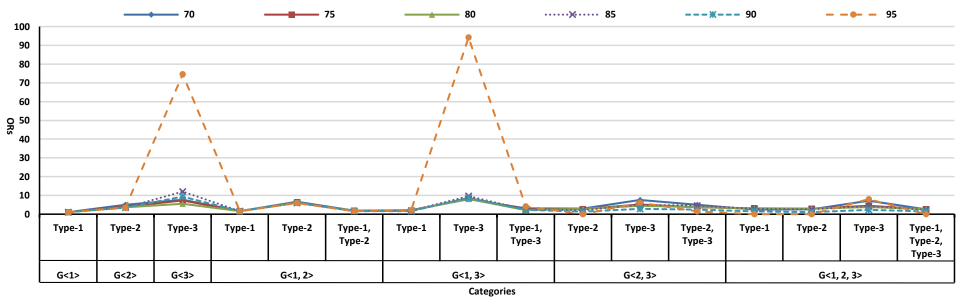
Genealogies G<1, 3>: Similar to G<1, 2>, ORs for genealogies containing Type-1 and Type-3 clones are in general higher than ORs for genealogies containing either Type-1 or Type-3 clones only; implying that the mutation of a clone from Type-1 to Type-3 or conversely increases the risk for faults. G<1, 3> genealogies predominated by Type-3 clones are more risky. For JBOSS and ARGOUML, when Type-1 and Type-3 clones are equally frequent in a G<1, 3> genealogy, this risk for fault is reduced. This finding is confirmed by the sensitivity analysis for all cases, but the case of ARGOUML, when Type-3 clones are detected using either 70% or 75% similarity threshold.

Genealogies G<2, 3>: For JBOSS and ARGOUML, ORs for genealogies containing Type-2 and Type-3 clones are in general higher than ORs for genealogies containing either Type-2 or Type-3 clones only, suggesting that in these two systems, mutations between Type-2 and Type-3 increase the risk for fault. In JBOSS and ARGOUML, G<2, 3> genealogies predominated by Type-2 clones are the most risky. For APACHE-ANT, mutations of clones from Type-2 to Type-3 or conversely, reduced the risk for fault. G<2, 3> genealogies predominated by Type-2 are the less risky for APACHE-ANT. These results are confirmed by the sensitivity analysis, with the exception of JBOSS and ARGOUML, when Type-3 clones are detected with the 95% similarity threshold.

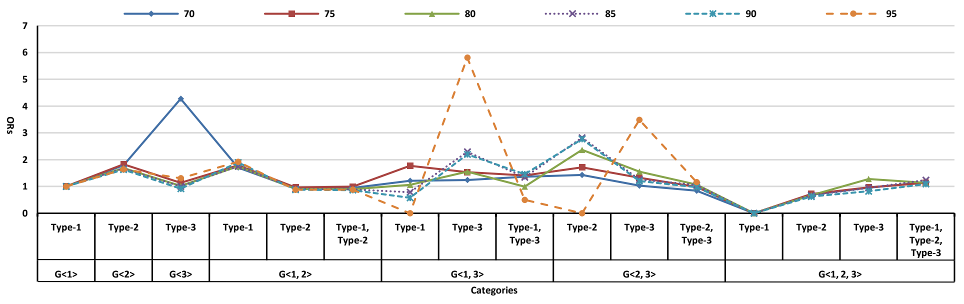
Genealogies G<1, 2, 3>: In genealogies containing Type-1, Type-2, and Type-3 clones, when Type-1, Type-2, and Type-3 clones are equally frequent, the risk for faults is reduced. G<1, 2, 3> genealogies that are predominated by Type-3 clones are the most risky. The sensitivity analysis confirms this result for APACHE ANT. For ARGOUML, G<1, 2, 3> genealogies where Type-1, Type-2, and Type-3 clones are equally frequent



(a) JBoss



(b) Apache Ant



(c) ArgoUML

Figure 4. Sensitivity Analysis for Clone Mutation in JBoss, Apache-Ant, and ArgoUML

are the most risky, when Type-3 clones are detected using either 70%, 75%, 85%, 90% or 95% similarity threshold.

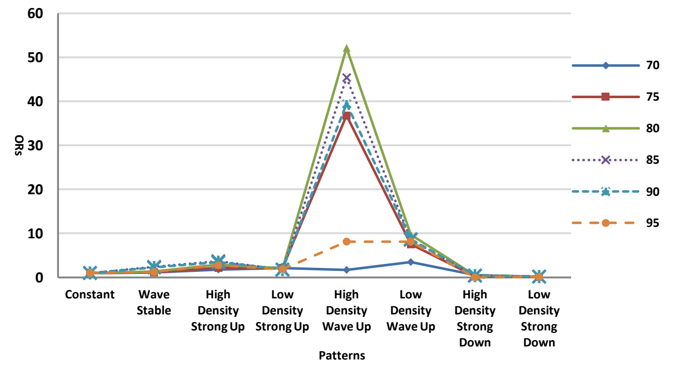
To examine the potential effect of LOC (line of code) on our results, we computed and compared the LOC of all clones (*i.e.*, Type-1, Type-2, and Type-3 clones) contained in the genealogies extracted from our three subject systems. We observed that, except in the case of JBOSS, where Type-1 clones are smaller than Type-2 and Type-3 clones, in general, there is no significant difference between the sizes of the clones. Consequently, we conclude that size alone cannot explain the results obtained in Table VII and Figure 4.

Overall, we conclude that the mutation of clone groups to Type-2 or Type-3 clones increases the risk for fault.

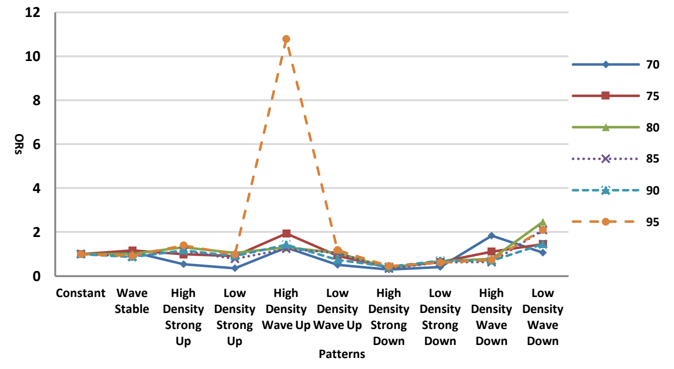
RQ3: Are some clone migrations more fault-prone than others?

Motivation. When performing modifications on cloned parts of a software system, developers should be aware of all the code segments involved in the clone groups. A developer overlooking a cloned code segment is at risk of introducing a fault in the software system. The purpose of this research question is to investigate the potential impact on fault-proneness of the migration of cloned code segments across the directories of a software system. The clones in a group can be changed, so the directories of those clones in that group can also be changed. More specifically, we want to verify using the ten migration patterns described in Table II, if some displacements of cloned code segments during maintenance and evolution activities are likely to increase the risk for faults in a system.

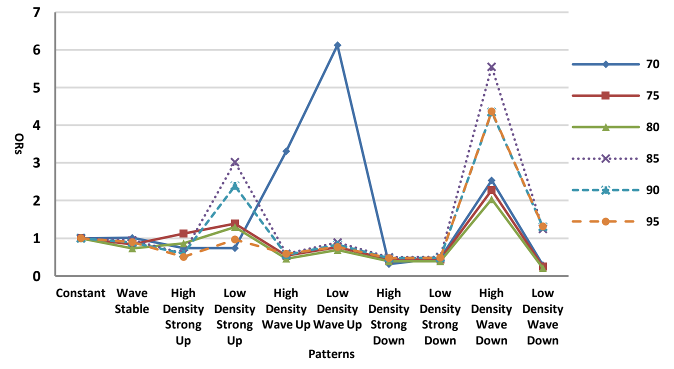
Approach. For each clone group G, from the clone genealo-



(a) JBoss



(b) Apache Ant



(c) ArgoUML

Figure 5. Sensitivity Analysis for Clone Migrations in JBoss, Apache-Ant, and ArgoUML

gies extracted in Section V, and for each two code segments A, B in G, we compute the distance $d_{dir}(A, B)$ between A and B following the definition presented in Section III. We also compute the size of G. Based on the variation of the size of G and the variation of the median distance between the code segments in G, we identify the migration patterns (*e.g.*, *High Density Strong Up*) of all the genealogies using the criteria described in Table II. We classify the clone genealogies according to the migration patterns.

For each migration pattern, we compute the number of fault-containing and fault-free genealogies and formulate the following null hypothesis: H_{03}*: the proportion of clone groups that experience fault fixes is the same for all the ten clone migration patterns*. We use the Chi-square test and compute Odds ratio to test H_{03}. When computing Odds ratios, we select the *Constant* migration pattern as the control group. We form one experimental group for each of the remaining nine migration patterns. We perform the Chi-square test using the 5% level. We also perform a sensitivity analysis to assess the effect that a similarity threshold used to detect Type-3 clones can have on our results.

Findings. Table VIII shows the result of the Chi-square test and lists *ORs* for the ten migration patterns described in Table II. The results are statistically significant for all three systems, *i.e.*, APACHE-ANT, ARGOUML and JBOSS. Overall, we can reject H_{03}.

High Density Strong Up, Low Density Strong Up, High Density Wave Up, and Low Density Wave Up patterns are more fault-prone than the Constant pattern in JBOSS and APACHE-ANT. In ARGOUML, the Low Density Strong Up pattern is also more fault-prone than the Constant pattern. These results suggest that when the distance between the code segments in a clone group is increased, the risk for fault increase. Dispersing and regrouping duplicated code segments (*i.e.*, Wave) is also risky, as suggested by the ORs of the High Density Wave Down pattern in ARGOUML (*i.e.*, 2.03) and the OR of the Low Density Wave Down pattern in JBOSS (*i.e.*, 2.14) and APACHE-ANT (*i.e.*, 2.47). Globally, these findings are corroborated by the results of the sensitivity analysis illustrated on Figure 5. If possible, development teams should avoid changing the location of cloned code during maintenance and evolution activities.

In sum, we conclude that clone migration is risky. Increasing the distance between code segments in a clone group increases the risk for fault.

VI. THREATS TO VALIDITY

In this section, we discuss the threats to validity of our study following common guidelines [17] of empirical studies.

Construct validity threats concern the relation between theory and observation. In our study these threats are mainly from the reliability of the tools used for clone detection. To reduce the possibility of misclassifying code segments as clones, we choose to use a mature clone detection tool that has been used in previous studies (*i.e.*, NICAD). NICAD can detect both exact and near-miss clones with high precision and recall [14].

Another treat to construct validity concerns the accuracy of J-REX which uses the same algorithm as previous studies by Hassan *et al.* [18] and Mockus *et al.* [19]. Hassan [12] has compared a classification of commit messages based on the algorithm to a manual evaluation of commit messages by six professional developers from the industry and found a correlation $\sigma > 0.8$. The ability of J-REX to recognize fault fixes is then comparable to that of a professional developer.

Threats to internal validity do not affect this study, as it is an exploratory study [17]. We cannot claim causation, we simply report observations and correlations, although we try to explain these observations in our discussions.

Conclusion validity threats concern the relation between the treatment and the outcome. We are careful to acknowledge the assumptions of each statistical test used in our study. We used non-parametric tests that do not require making assumptions about the data set distribution.

Reliability validity threats concern the possibility of replicating this study. We provide all necessary details needed to replicate our study. All our three subject systems are publicly available for study.

Threats to *external validity* concern the possibility to generalize our results. We examine three large Java open source software systems. Both of them use a plug-in architecture. However they are from different domains and have different sizes. More studies on other systems are necessary to further validate our findings.

VII. Conclusion

In this study, we examine the *mutation* and *migration* of clone groups in software systems. We identify seven categories of clone genealogies and ten clone migration patterns, which we study in detail to identify the most frequent clone *mutation* and clone *migration*. We also investigate the fault-proneness of the different clone genealogies and clone migration patterns to identify the most risky types of clone *mutation* and clone *migration*. Results show that the most frequent form of clone *mutation* is between Type-1 and Type-2 clones. The most frequent migration pattern is the *Wave Stable* pattern, which is a pattern where the distance between the code segments in the clone group increases and decreases by the same amount throughout the evolution of the software system.

When clone groups are mutated to either Type-2 or Type-3 clones, the risk for faults is increased. Clone genealogies predominated by Type-2 clones are generally most fault-prone. However, when all the clone types in a clone genealogy are equally frequent, the risk for faults is reduced.

Clone groups involved in migration patterns characterized by an increase of the distance between cloned code segments are more prone to faults than others. Globally, a modification of the location of cloned code segments during the evolution of a software system increases the risk for faults in the system.

From this study, we can conclude that the different types of clone *mutation* and clone *migration* are inconsistently risky. Only some clone *mutations* and *migrations* require monitoring. Especially, development teams should be careful when mutating Type-1 clones to either Type-2 or Type-3 clones. Also, special attention should be granted to clone groups where the location of code segments is modified during a revision of the system. In the future, we plan to expand this study to include more software systems written in other programming languages. We also plan to use other clone detection tools to further validate our findings.

References

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Software Eng.*, pp. 577–591, 2007.

[2] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 187–196.

[3] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, pp. 1–34, 2010.

[4] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, sept. 2011, pp. 273 – 282.

[5] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," 2007.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654 – 670, jul 2002.

[7] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *ICSM'11*, 2011, pp. 273–282.

[8] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 81 –90.

[9] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and forecasting near-miss clones in evolving software: An empirical study," in *ICECCS'11*, 2011, pp. 295–304.

[10] W. Shang, Z. M. Jiang, B. Adams, and A. Hassan, "Mapreduce as a general framework to support research in mining software repositories (msr)," in *6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 21 –30.

[11] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings. International Conference on Software Maintenance*, 2000.

[12] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[13] N. Göde and J. Harder, "Clone stability," in *15th European Conference on Software Maintenance and Reengineering*, May 2011.

[14] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC'08*, 2008, pp. 172–181.

[15] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," *Working Conference on Reverse Engineering*, pp. 13–21, 2010.

[16] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[17] R. K. Yin, "Design and methods third edition, 3rd ed." in *ICSM'00*, 2002.

[18] A. E. Hassan and R. C. Holt, "Studying the evolution of software systems using evolutionary code extractors," in *IWPSE'04*, 2004, pp. 76–81.

[19] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM'00*, 2000, pp. 120–130.