2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

When does a Refactoring Induce Bugs? An Empirical Study

Gabriele Bavota[1], Bernardino De Carluccio[1], Andrea De Lucia[1]
Massimiliano Di Penta[2], Rocco Oliveto[3], Orazio Strollo[1]
[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
gbavota@unisa.it, bernardino.decarluccio@gmail.com, adelucia@unisa.it
dipenta@unisannio.it, rocco.oliveto@unimol.it, oraziostrollo@hotmail.com

***Abstract*—Refactorings are—as defined by Fowler—behavior preserving source code transformations. Their main purpose is to improve maintainability or comprehensibility, or also reduce the code footprint if needed. In principle, refactorings are defined as simple operations so that are "unlikely to go wrong" and introduce faults. In practice, refactoring activities could have their risks, as other changes.**

This paper reports an empirical study carried out on three Java software systems, namely Apache Ant, Xerces, and ArgoUML, aimed at investigating to what extent refactoring activities induce faults. Specifically, we automatically detect (and then manually validate) 15,008 refactoring operations (of 52 different kinds) using an existing tool (Ref-Finder). Then, we use the SZZ algorithm to determine whether it is likely that refactorings induced a fault.

Results indicate that, while some kinds of refactorings are unlikely to be harmful, others, such as refactorings involving hierarchies (e.g., pull up method), tend to induce faults very frequently. This suggests more accurate code inspection or testing activities when such specific refactorings are performed.

***Index Terms*—Refactoring, Fault-inducing changes, Mining software repositories, Empirical Studies.**

I. Introduction

Software systems are continuously subject to maintenance tasks to introduce new features or fix bugs [1]. Very often such activities are performed in an undisciplined manner due to strict time constraints, to lack of resources/skills, or to the limited knowledge some developers have of the system design [2]. As a result, the code underlying structure, and therefore the related design, tend to deteriorate.

This phenomenon was defined as "*software aging*" by Parnas [3], and was also described in the law of increasing complexity by Lehman [1]. Some researchers measured the phenomenon in terms of change entropy [4], [5], while others defined "*antipatterns*", i.e., recurring cases of poor design choices occurring as a consequence of aging, or when the software is not properly designed from the beginning. Classes doing too much (*God classes* or *Blobs*), poorly structured code (*Spaghetti code*), or *Long Message Chains* used to develop a certain feature are only few examples of antipatterns that plague software systems [2].

In order to mitigate the above described issues, software systems are, time to time, subject to improvement activities, aimed at enhancing the code and design structure. Such activities are often referred to as *refactoring*. Refactoring is defined by Fowler [2] as "*a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior*". The aim of refactoring is to improve the structure of source code—and consequently of the system design—whenever its structure may possibly lead to maintainability or comprehensibility problems. Fowler's catalogue [6] comprises a set of 93 refactorings, aimed at dealing with different antipatterns in source code, such as, extracting a class from a Blob, pulling up a method from a subclass onto a superclass, or modifying the navigability of an association between two classes.

In theory, a refactoring should not change the behavior of a software system, but only help in improving some of its non-functional attributes. In practice, a refactoring might be risky as any other change occurring in a system, causing possible bug introductions. Indeed, a recent study [10] showed that even automated refactoring as performed by Integrated Development Environments could be fault-prone as well.

While there are attempts to investigate the relation between some refactorings and fault-proneness [8], [9] or change entropy [7], to the best of our knowledge there is no study aimed at thoroughly investigating whether a wide set of (even undocumented) refactorings occurred in a software system during its evolution induced bugs, and what kind of refactorings might induce more bugs than others.

In this paper we report an empirical study aimed at investigating to what extent refactoring induces bug fixes in software systems. We use an existing tool, namely Ref-Finder [11], to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems, Apache Ant[1], ArgoUML[2], and Xerces-J[3]. Of the 15,008 refactoring operations detected by the tool, 12,922 operations have been manually validated as actually refactorings. Then, we use the SZZ algorithm [12], [13] to determine whether the 12,922

[1]http://ant.apache.org/
[2]http://argouml.tigris.org/
[3]http://xerces.apache.org/xerces-j/

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.20

refactoring operations induced[4] bug fixes.

Specifically, the paper aims at investigating

1) to what extent refactorings, in general, induce bugs fixes, i.e., what is the percentage of refactoring activities that likely induced a bug fixing;
2) whether specific kinds of refactorings induce more bugs than others;
3) whether bugs occurred more in the source or target source code components involved in refactorings at least for refactorings that—as *Move Method* for example—involve more components.

Results show that while, in general, the percentage of bug fixes likely induced by refactorings is relatively low (i.e., 15%), there are some specific kinds of refactorings that are very likely to induce fixes. In particular, *Pull Up Method* and *Extract Subclass* (two refactoring operations related to changes applied to the class hierarchy) induce (in percentage) more fixes than the others. Moreover, bugs generally occur more in the target source code components involved in refactorings.

Structure of the paper. Section II describes the approach followed to extract the data necessary for this study. The empirical study is then described in Section III, while results are presented in Section IV. Section V discusses the threats to validity of our study. Finally, Section VII concludes the paper, after a discussion of related work (Section VI).

II. Data Extraction Process

This section describes the data extraction process we followed with the aim of (i) identifying refactoring performed across system versions and (ii) determine whether refactorings could have likely induced a fix.

A. Step 1: Identifying Refactorings

We use Ref-Finder [11] to detect refactorings performed between each subsequent couples of releases of the systems under analysis. We decided to work at release level for convenience and because the current implementation of Ref-Finder identifies refactoring operations between two releases of a software system. Ref-Finder has been implemented as an Eclipse plug-in and it is able to detect 63 different kinds of refactorings.

In a case study conducted on three open source systems Ref-Finder was able to detect refactoring operations with an average recall of 95% and an average precision of 79% [11]. Even if the accuracy of such a tool is quite high, we tried to (at least) mitigate problems related to false positives (precision) through manual validation of the refactorings proposed by Ref-Finder. Specifically, each refactoring operation identified by the tool was manually analyzed through source code inspection by two Master students from the University of Salerno. The students individually validated each of the proposed refactoring operations. For the study reported in this paper, the manual validation required about 10 working days of two persons.

Once students validated the refactorings, they performed an open-discussion with two of the authors of this paper to solve conflicts and reach a consensus on the refactoring operations analyzed, classifying them as *true positive* or *false positive*. The output of this stage is a set of triples (rel_j, ref_k, C), where rel_j indicates the release number, ref_k the kind of refactoring occurred, and C is the set of refactored classes. Each release is also associated to the release date.

B. Step 2: Identifying Post-release Bugs

In order to identify all post release bugs, we first extract the change log from the versioning system (two of the systems used Git[5], while for the third one, ArgoUML, we converted the SVN repository into a Git repository). Then, we identify bug fixing changes by mining regular expressions containing issue IDs in the versioning system change log, e.g., *"fixed issue #ID"* or *"issue ID"*. After that, for each issue ID identified, we download the corresponding issue report from the issue tracking system Bugzilla[6] or Jira[7] (depending on the system), and extract from it

- *product name*;
- *issue type*, i.e., whether the issue is a bug or a request for enhancements or other kinds of changes. In some versions of Bugzilla this classification is (partially) handled by the field *severity*;
- *issue status*, i.e., whether the issue was closed or not;
- *issue resolution*, i.e., whether the issue was resolved by fixing it, or whether it was a duplicate issue, or a "works for me" case;
- *issue opening date*;
- *issue closing date*, if any.

Then, we check that the issue report was correctly downloaded (e.g., the issue ID identified from the versioning system commit note could be a false positive), and that the issue, indeed, is related to the product under analysis (e.g., Apache uses the same issue tracking system for multiple products). After that, we use the issue type/severity field to classify the issue and distinguish bug fixing from other issues (e.g., enhancements). Finally, we only consider bugs having the *CLOSED* status and the *FIXED* resolution. Basically, we restrict our attention to (i) issues that were related to bugs as we use them as a measure of fault-proneness, and (ii) issues that were not duplicate nor false alarms.

C. Step 3: Bug-inducing changes

The third step of the approach aims at identifying changes that likely induced the bug. We use the SZZ algorithm [12], [13], which relies on the annotation/blame feature of versioning systems. In essence, given a bug fix identified by the bug ID k, the approach works as follows

[4]In this paper we use the term "inducing" used by the authors of the SZZ algorithm. By that, it is meant that a change induces a bug fixing if it last modifies a source code line involved in a bug fixing. This does not necessarily mean a causation between a change and a bug fixing.

[5]http://git-scm.com/

[6]http://www.bugzilla.org/

[7]http://www.atlassian.com/software/jira/overview

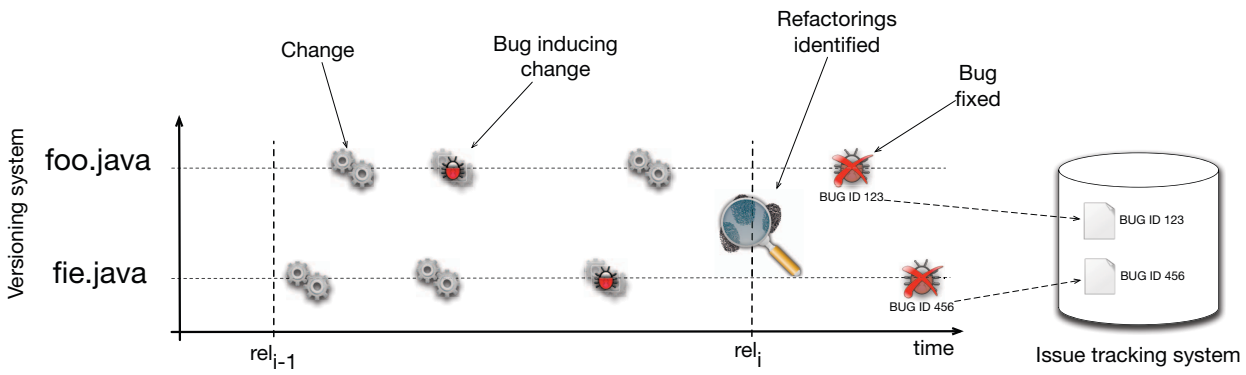

Fig. 1. The approach used to identify whether a refactoring induces a bug.

1) For each file f_i, $i = 1 \dots m_k$ involved in the bug fix k (m_k is the number of files changed in the bug fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2) Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of Git is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island parser. This produces, for each file f_i, a set of $n_{i,k}$ bug-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$.

D. Step 4: When does a refactoring induce a bug fixing?

Figure 1 summarizes how, based on the information extracted in the previous steps, we assume that a refactoring activity can likely induce a bug fixing. Specifically

- a refactoring *ref*$_k$ occurred on class c_j on *rel*$_i$;
- a bug-inducing change for class c_j was performed between *rel*$_{i-1}$ and *rel*$_i$;
- the bug inducing change resulted in a bug report opened after *rel*$_i$.

III. Empirical Study Design

The *goal* of the study is to analyze refactoring operations occurring over the history of a software project, with the *purpose* of understanding to what extent they could induce bug fixes. The *quality focus* is software maintainability and comprehensibility, which could be improved by means of refactoring, but also software fault-proneness, which could be influenced by refactoring activities. The *perspective* is of researchers interested in investigating whether source code subject to refactoring—and in particular to some specific kinds of refactorings—deserves particular attention during Verification & Validation activities.

A. Context and Research Questions

The *context* of the study consists of 63 releases of three Java open source projects, namely Apache Ant, ArgoUML, and Xerces-J. Apache Ant is a build tool and library specifically conceived for Java applications (though it can be used for other purposes). ArgoUML is an open source UML modeler. Xerces-J is a XML parser for Java. Table I reports characteristics of the analyzed systems, namely versions analyzed, and size range (in terms of KLOC and # of classes). The table also reports the number of refactoring operations (as well as the number of different kinds of refactorings) identified on the three systems after the manual validation of the refactorings identified by Ref-Finder.

In the context of the study, we formulated the following research questions:

- **RQ$_1$:** *To what extent do refactorings induce bug fixes?* This research question aims at investigating whether classes subjects to refactoring underwent more bug-fix inducing changes than others. The rationale is to investigate whether refactoring induces bug-fixes more than other kinds of changes, or whether to this extent there is no difference with respect to other changes. That is, refactoring could still induce bug fixes, however it must be checked whether this happens more frequently than for other changes. Specifically, we test the null hypothesis:

  H_{0_1}: There is no significant difference in proportions of classes subject to bug-fix inducing changes between classes subject or not to refactorings.
- **RQ$_2$:** *How do various refactorings differ in terms of proneness to induce bug fixes?* This research question investigates whether different kinds of refactorings induce bug fixes in different proportions. The rationale is to investigate whether some kinds of refactorings may be more fault-prone than other, so that better Verification & Validation activities are needed when such specific refactorings are performed. The null hypothesis being tested is:

  H_{0_2}: There is no significant difference among proportions of bug-fix inducing refactorings among different kinds of refactorings.
- **RQ$_3$:** *Are refactorings more likely to induce bug fixes in source or target components?* Since some specific refactorings involve a source and a target source code components (e.g., when moving a method from a class

TABLE I
CHARACTERISTICS OF THE ANALYZED PROJECTS

Project	Period	Analyzed Releases	(Num)	Classes	KLOC	Refactorings	Kinds of Refactorings
Apache Ant	Jan 2000-Dec 2010	1.2-1.8.2	17	87-1,191	8-255	1,469	31
ArgoUML	Oct 2002-Dec 2011	0.12-0.34	13	777-1,519	362-918	3,532	43
Xerces	Nov 1999-Nov 2010	1.0.4-2.9.1	33	181-776	56-179	7,921	43

to another), it would be worthwhile to investigate whether refactorings tend to induce bug fixes more frequently in the source or in the target. Bugs occurring in the source could be for example due to the code moved away from there (e.g., missing behavior after refactoring occurred; or, in case an overriding method is moved away from a class, there is risk that other methods might use the method from the superclass). Bugs occurring in the target might be due to wrong assumptions the moved method is doing when using resources of the target component. The null hypothesis being tested is the following:

H_{0_3}: There is no significant difference among proportions of bug-fix inducing refactorings among source and target code involved in refactorings.

B. Analysis Method

This subsection describes the analyses and statistical procedures used to address the three research questions formulated in Section III-A. All statistical tests assume a significance level of 95%.

To address $\mathbf{RQ}_1$, for each release we compute:

- **NB-NR**, the number of classes of a project release i for which there was no bug-fix inducing change nor any refactoring between release $i-1$ and release i;
- **B-NR**, the number of classes of a project release i for which there was at least one bug-fix inducing change, but no refactoring, between release $i-1$ and release i;
- **NB-R**, the number of classes of a project release i for which there was no bug-fix inducing change, while there was at least one refactoring between release $i-1$ and release i; and
- **B-R**, the number of classes of a project release i for which there was at least one bug-fix inducing change and at least one refactoring between release $i-1$ and release i.

Then, we use the Fisher's exact test [14] to test whether the proportion between (**B-NR**, **NB-NR**) and between (**B-R**, **NB-R**) significantly differ.

In addition, we use the Odds Ratio (OR) [14] as effect size measure. OR is defined as the ratio of the odds of an event occurring in one group to the odds of it occurring in another group. For contingency matrices, like our case, the OR is defined as: $OR = (\text{B-NR}/\text{NB-NR}) \,/\, (\text{B-R}/\text{NB-R})$. An OR of 1 indicates that the condition or event under study is equally likely in both groups. An OR greater than 1 indicates that the condition or event is more likely in the first group. Vice versa, an OR lower than 1 indicates that the condition or event is more likely in the second group.

As for $\mathbf{RQ}_2$, we report and compare, for different kinds of refactorings, the overall number of classes involved in different kinds of refactorings performed, and the number and percentage of these classes for which refactorings likely induced a bug fix. Then, we statistically compare such proportions using proportion test (across all kinds of refactorings), and then we pairwise compare refactorings using Fisher's exact test. We also compute the OR effect size, which indicates, in this case, the chances that a specific kind of refactoring induced a bug fixing in a class as opposed to another kind of refactoring. Since we perform several tests on the same data, we adjust p-values using the Holm's correction procedure [14]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

Finally, to address $\mathbf{RQ}_3$, we report and compare the overall number and percentage of refactorings that (i) induced a bug fixing in the source class and that (ii) induced a bug in the target class. We perform a statistical comparison as done for $\mathbf{RQ}_2$.

TABLE II
ANT: NUMBER OF CLASSES INVOLVED (OR NOT) IN REFACTORING AND IN BUG-INDUCING CHANGES, AND RESULTS OF FISHER'S EXACT TEST.

RELEASE	NB-NR	NB-R	B-NR	B-R	P-VALUE	OR
1.2	153	18	0	0	1.00	0.00
1.3	375	10	76	5	0.15	2.46
1.4	398	27	0	0	1.00	0.00
1.4.1	424	1	0	0	1.00	0.00
1.5	659	59	139	16	0.43	1.29
1.5.1	718	4	0	0	1.00	0.00
1.5.2	719	8	81	4	**0.03**	4.42
1.6.0	852	62	0	0	1.00	0.00
1.6.1	907	3	39	1	0.16	7.71
1.6.2	918	11	0	0	1.00	0.00
1.6.3	937	18	39	5	$<$ **0.01**	6.65
1.6.4	954	1	0	0	1.00	0.00
1.7.0	1,053	60	30	6	**0.01**	3.50
1.7.1	1,077	46	37	7	$<$ **0.01**	4.42
1.8.0	1,122	50	24	2	0.31	1.87
1.8.1	1,159	17	0	0	1.00	0.00
1.8.2	1,181	10	0	0	1.00	0.00

IV. EMPIRICAL STUDY RESULTS

This section discusses the results achieved in our study aiming at responding to our research questions. Working data sets are available for replication purposes[8].

A. RQ_1: To what extent do refactorings induce bug fixes?

Tables II, III, and IV, report—for Ant, ArgoUML, and Xerces, respectively–the number of classes that, for each release, belong to the four categories (NB-NR, NB-R, B-NR,

[8]http://www.distat.unimol.it/reports/refactoring-defect/

TABLE III
ARGOUML: NUMBER OF CLASSES INVOLVED (OR NOT) IN REFACTORING AND IN BUG-INDUCING CHANGES, AND RESULTS OF FISHER'S EXACT TEST.

RELEASE	NB-NR	NB-R	B-NR	B-R	P-VALUE	OR
0.12	791	59	165	6	0.12	0.49
0.14	1,036	97	0	0	1.00	0.00
0.18.1	913	252	313	84	0.89	0.97
0.20	1,101	174	119	22	0.52	1.17
0.22	1,126	171	0	0	1.00	0.00
0.24	3,850	112	93	14	$<$ **0.01**	5.17
0.26	1,187	317	99	30	0.58	1.13
0.28	1,438	64	0	0	1.00	0.00
0.28.1	1,500	2	52	0	1.00	0.00
0.30	1,461	58	2	0	1.00	0.00
0.30.1	1,515	4	0	0	1.00	0.00
0.32	1,465	32	0	0	1.00	0.00
0.34	1,219	15	0	0	1.00	0.00

TABLE IV
XERCES: NUMBER OF CLASSES INVOLVED (OR NOT) IN REFACTORING AND IN BUG-INDUCING CHANGES, AND RESULTS OF FISHER'S EXACT TEST.

RELEASE	NB-NR	NB-R	B-NR	B-R	P-VALUE	OR
1.0.4	337	89	3	3	0.11	3.77
1.2.0	388	75	0	0	1.00	0.00
1.2.1	441	20	0	0	1.00	0.00
1.2.2	443	19	0	0	1.00	0.00
1.2.3	460	2	0	0	1.00	0.00
1.3.0	443	34	2	2	**0.03**	12.86
1.3.1	454	24	0	0	1.00	0.00
1.4.0	485	10	0	0	1.00	0.00
1.4.1	491	4	0	0	1.00	0.00
1.4.2	492	10	0	0	1.00	0.00
1.4.3	497	1	0	0	1.00	0.00
1.4.4	481	18	0	0	1.00	0.00
2.0.1	657	1	0	0	1.00	0.00
2.0.2	564	151	0	0	1.00	0.00
2.1.0	708	3	0	0	1.00	0.00
2.2.0	686	36	0	0	1.00	0.00
2.2.1	700	23	0	0	1.00	0.00
2.3.0	688	88	17	22	$<$ **0.01**	10.06
2.4.0	614	41	0	0	1.00	0.00
2.5.0	592	73	0	0	1.00	0.00
2.6.0	555	119	0	0	1.00	0.00
2.6.1	644	32	11	10	$<$ **0.01**	18.09
2.6.2	676	19	0	0	1.00	0.00
2.7.0	566	140	4	16	$<$ **0.01**	16.09
2.7.1	700	6	0	0	1.00	0.00
2.8.0	664	45	10	6	$<$ **0.01**	8.79
2.8.1	688	22	0	0	1.00	0.00
2.9.0	680	32	0	0	1.00	0.00
2.9.1	714	2	2	1	**0.01**	157.69

and B-R) described in Section III-B. Also, the tables report results of the Fisher's exact test (significant p-values are shown in bold face) and ORs.

For Ant, in most cases there is no significant difference in proportions, that is, the proportion of bug-fix inducing changes is similar for classes involved or not in refactorings. There are four releases (1.5.2, 1.6.3, 1.7.0, and 1.7.1) for which differences are significant. In all these cases, the OR is high, i.e., the chances for a class involved in refactorings to undergo bug-fix inducing changes is from 3.5 to 6.6 times higher than for other classes. Noticeably, this happened for releases—such as 1.7.0 or 1.7.1—where the number of performed refactorings is particularly high (although other releases with a high number of refactorings such as 1.5 and 1.8.0 did not exhibit a similar behavior). Note that releases 1.7.0 and 1.7.1 are the ones were we found the higher number of refactorings in Ant, i.e., 368 and 225, respectively. Moreover, manual analysis of the versioning system logs between versions 1.6.4 and 1.7.0 and between 1.7.0 and 1.7.1 confirmed as often developers talked about the performed refactoring operations.

For ArgoUML, there is only one release (0.24) where differences are significant. To understand the reason behind this result, also in this case we analyzed the versioning system logs between versions 0.22 and 0.24. We found that very often changes were related to refactoring activities (almost 100 of developers' comments explicitly described the performed refactoring operations). Thus, this ArgoUML version was strongly subject to refactoring activities (as also confirmed by the 401 refactoring operations we found among these system's versions). It can be noticed that the number of classes involved in bug-fix inducing changes is very low in the last few versions because these versions are relatively recent and thus possible introduced bugs have still not been reported.

For Xerces, out of the 22 releases analyzed, the differences are significant only in 7 cases. In all of them the ORs are very high, highlighting again that at least for all releases where the difference is significant, classes involved in refactorings have a higher chance of being involved in bug-fix inducing changes.

Summary for RQ$_1$: In general, there is no significant difference in the proportion of classes involved in bug-fix inducing changes between classes involved or not in refactorings. Such differences occur in some specific cases, and in all of them classes involved in refactorings have a higher chance of being involved in bug-inducing changes.

B. RQ$_2$: How do various refactorings differ in terms of proneness to induce bug fixes?

In the analyzed 63 releases of the three object systems we found 52 kinds of refactoring operations applied. Table V shows, for each of them, (i) the number of classes involved by the refactoring (column #Ref. Cl.), (ii) the number of faulty classes among the refactored ones (column #Faulty Cl.), and (iii) the proportion of refactored classes for which the refactoring likely induced a bug fixing (column Prop.). We report separate results for each system as well as aggregate results (in the rightmost column). Overall, analyzing all kinds of refactoring operations on all the object systems, we found that 1,616 refactored classes out of the 10,969 identified (15%) have been subject to bug fixing.

We identified—across the analyzed systems' releases—a total of 12,922 refactoring operations (manually validated among the 15,008 retrieved by Ref-Finder). It is worth noting that among these 9,816 are related to only 8 kinds

TABLE V
REFACTORED CLASSES AND FAULT-PRONE CLASSES (AMONG THOSE SUBJECT TO REFACTORING).

Operation	Ant			ArgoUML			Xerces			Total		
	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.	#Ref. Cl.	#Faulty Cl.	Prop.
Add Parameter	133	33	25%	511	65	13%	665	73	11%	1,309	171	13%
Change Bidirectional Association to Unidirectional	0	0	-	2	1	50%	3	1	33%	5	2	40%
Change Unidirectional Association to Bidirectional	0	0	-	0	0	-	6	3	50%	6	3	50%
Collapse Hierarchy	0	0	-	1	0	0%	3	1	33%	4	1	25%
Consolidate Conditional Expression	32	5	16%	45	5	11%	171	34	20%	248	44	18%
Consolidate Duplicate Conditional Fragments	73	15	21%	103	21	20%	422	28	7%	598	64	11%
Decompose Conditional	1	0	0%	2	2	100%	0	0	-	3	2	67%
Encapsulate Field	0	0	-	1	0	0%	0	0	-	1	0	0%
Extract Hierarchy	0	0	-	4	2	50%	3	0	0%	7	2	29%
Extract Interface	10	0	0%	40	4	10%	78	1	1%	128	5	4%
Extract Method	73	19	26%	135	40	30%	166	20	12%	374	79	21%
Extract Subclass	0	0	-	4	2	50%	6	2	33%	10	4	40%
Extract Superclass	3	0	0%	13	1	8%	2	0	0%	18	1	6%
Form Template Method	0	0	-	10	0	0%	0	0	-	10	0	0%
Hide Delegate	0	0	-	0	0	-	1	0	0%	1	0	0%
Hide Method	0	0	-	9	5	56%	0	0	-	9	5	56%
Inline Class	1	1	100%	0	0	-	1	0	0%	2	1	50%
Inline Method	25	2	8%	22	5	23%	74	11	15%	121	18	15%
Inline Temp	55	27	49%	98	28	29%	86	8	9%	239	63	26%
Introduce Assertion	23	0	0%	14	1	7%	0	0	-	37	1	3%
Introduce Explaining Variable	115	10	9%	104	30	29%	165	16	10%	384	56	15%
Introduce Local Extension	3	0	0%	18	0	0%	25	0	0%	46	0	0%
Introduce Null Object	2	0	0%	25	10	40%	35	2	6%	62	12	19%
Introduce Parameter Object	0	0	-	0	0	-	16	0	0%	16	0	0%
Move Field	67	10	15%	399	111	28%	920	41	4%	1,386	162	12%
Move Method	96	9	9%	349	76	22%	747	66	9%	1,192	151	13%
Parameterize Method	1	1	100%	1	1	100%	2	0	0%	4	2	50%
Preserve Whole Object	0	0	-	0	0	-	3	0	0%	3	0	0%
Pull Up Constructor Body	1	0	0%	5	0	0%	0	0	-	6	0	0%
Pull Up Field	2	0	0%	4	0%	0%	6	1	17%	12	1	8%
Pull Up Method	4	0	0%	0	0	-	11	6	55%	15	6	40%
Push Down Field	0	0	-	0	0	-	52	2	4%	52	2	4%
Push Down Method	0	0	-	1	0	0%	44	0	0%	45	0	0%
Remove Assignment To Parameters	49	6	12%	40	11	20%	73	9	12%	162	26	16%
Remove Control Flag	26	8	31%	147	34	23%	136	9	7%	309	51	17%
Remove Middle Man	0	0	-	1	0	0%	0	0	-	1	0	0%
Remove Parameter	114	33	29%	442	70	16%	496	61	12%	1,052	164	16%
Rename Method	182	37	20%	262	49	19%	714	82	11%	1,158	168	15%
Replace Conditional With Polymorphism	0	0	-	4	2	50%	6	0	0	10	2	20%
Replace Constructor With Factory Method	1	0	0%	5	2	40%	5	0	0%	11	2	18%
Replace Data With Object	6	1	17%	10	1	10%	32	1	3%	48	3	6%
Replace Delegation With Inheritance	0	0	-	0	0	-	1	0	0%	1	0	0%
Replace Error Code With Exception	0	0	-	0	0	-	1	0	0%	1	0	0%
Replace Exception With Test	18	6	33%	19	2	11%	38	2	5%	75	10	13%
Replace Magic Number With Constant	327	53	16%	158	19	12%	516	85	16%	1,001	157	16%
Replace Method With Method Object	36	0	0%	374	115	31%	170	32	19%	580	147	25%
Replace Nested Conditional with Guard Clauses	13	0	0%	33	9	27%	124	14	11%	170	23	14%
Replace Parameter with Explicit Methods	0	0	-	1	1	100%	3	0	0%	4	1	25%
Replace Parameter with Method	0	0	-	3	3	100%	0	0	-	3	3	0%
Replace Temp with Query	0	0	-	1	0	0%	0	0	-	1	0	0%
Self Encapsulate Field	0	0	-	2	0	0%	7	0	0%	9	0	0%
Separate Query From Modifier	1	1	100%	2	0	0%	17	0	0%	20	1	5%

of refactorings (i.e., *Add Parameter*, *Consolidate Duplicate Conditional Fragments*, *Move Field*, *Move Method*, *Remove Parameter*, *Rename Method*, *Replace Magic Number With Constant*, *Replace Method With Method Object*). On the other side, other operations like *Encapsulate Field*, *Hide Delegate*, *Remove Middle Man*, *Replace Delegation With Inheritance*, *Replace Error Code With Exception*, and *Replace Temp with Query* are less frequent.

Table VI shows the results of Fisher's test and OR for cases where the test indicated a significant difference. It is worth noting that for refactorings for which there were a small number of instances, it was not possible to observe significant differences.

For Ant, we found that different kinds of refactorings exhibit a significantly different proportion of classes (prop test returned a p-value < 0.001) for which the refactoring likely induced a bug fixing. Pairwise Fisher's exact tests revealed that, for instance, *Inline Temp* induced a significantly higher proportion (49%) of fixes than other refactorings, with ORs indicating that such proportions are 9 times higher than *Move Method*, 5 times higher than *Move Field*, and 4 times higher than *Rename Method*.

For ArgoUML, again we found significant differences among different kinds of refactorings. Specifically, *Replace Method with Method Object* has a significantly high proportion (31%) of classes for which the refactoring likely induced a fix than other classes of refactorings, 2.4 times higher than *Remove Parameters* and 3.3 times higher than *Replace Magic Number with Constant*. Also, *Move Field* has a significantly higher proportion (28%), for example 2 times higher than *Remove Parameter*, 2.6 times higher than *Add Parameter*, and 3 times higher than *Replace Magic Number with Constant*.

For Xerces, refactorings with very high proportions—e.g., *Pull up Method* (55%)—were also present in few instances. Among refactorings for which the number of instances is high enough, we found that, for example, *Consolidate Conditional Expression* has significantly higher proportions than other refactorings, e.g., 5 times higher than *Move Field*.

An interesting result is that the two refactoring operations inducing (in percentage) more bugs (i.e., *Pull Up Method* and *Extract Subclass*) are both related to changes applied to a class hierarchy. In particular, both these refactorings induce bugs in 40% of cases. As for the *Pull Up Method* refactoring, it is used when the same exact method is implemented in two subclasses inheriting from the same superclass. To avoid code duplication, using the *Pull Up Method* refactoring the method

TABLE VI
RQ2: PAIRWISE COMPARISONS OF KINDS OF REFACTORINGS FOR WHICH FISHER'S TEST INDICATED A SIGNIFICANT DIFFERENCE.

ANT			
ref. kind 1	ref. kind 2	OR	p-value
inline_temp	introduce_explaining_variable	9.95	<0.01
inline_temp	move_field	5.41	0.03
inline_temp	move_method	9.15	<0.01
inline_temp	remove_assignment_to_parameters	6.78	0.02
inline_temp	rename_method	3.75	0.03
inline_temp	replace_magic_number_with_constant	4.96	<0.01
remove_parameter	introduce_explaining_variable	4.16	0.04
ARGOUML			
ref. kind 1	ref. kind 2	OR	p-value
extract_method	add_parameter	2.86	0.01
replace_method_with_method_object	remove_parameter	2.38	<0.01
replace_method_with_method_object	replace_magic_number_with_constant	3.22	<0.01
replace_method_with_method_object	add_parameter	3.03	<0.01
move_field	add_parameter	2.63	0.00
move_field	remove_parameter	2.05	0.02
move_field	replace_magic_number_with_constant	2.81	0.05
XERCES			
ref. kind 1	ref. kind 2	OR	p-value
consolidate_cond_expression	consolidate_duplicate_cond_fragments	3.48	0.01
consolidate_cond_expression	extract_interface	18.99	0.02
consolidate_cond_expression	move_field	5.31	<0.01
pull_up_method	consolidate_duplicate_cond_fragments	16.67	0.05
pull_up_method	extract_interface	> 100	<0.01
pull_up_method	move_field	25	<0.01
pull_up_method	push_down_method	>100	0.01
replace_magic_number_with_constant	consolidate_duplicate_cond_fragments	2.78	<0.01
replace_magic_number_with_constant	extract_interface	14.29	0.04
replace_magic_number_with_constant	move_field	4.16	<0.01
replace_method_with_method_object	consolidate_duplicate_cond_fragments	3.22	0.03
replace_method_with_method_object	extract_interface	16.67	0.03
replace_method_with_method_object	move_field	5	<0.01
remove_parameter	move_field	3.03	<0.01
rename_method	move_field	2.78	<0.01
add_parameter	move_field	2.64	<0.01

is deleted from the subclasses and moved to their parent. As for the *Extract Subclass* refactoring, it is applied when a class C implements features used only for some of its instances. In this case, a subclass of C is created, moving the interested features from C to its new subclass.

An example of *Extract Subclass* refactoring inducing a bug fixing is the one occurred in ArgoUML between the versions 0.14 and 0.18.1 and involving the class *StylePanelFig*. This class implements the basic style panel to manage the figures, allowing the user to see and set the figures' common attributes, i.e., the boundaries box, line, and fill color. Until version 0.14, this class also managed the shadow settings for figures. However, developers felt that the shadow setting does not belong to the "common attributes" of a figure since it can be applied only on particular types of figures and, for this reason, in version 0.18.1 they decided to move the shadow management responsibility in a new subclass of *StylePanelFig* called *StylePanelFigNodeModelElement*. However, after the subclass refactoring, the changes applied to the system introduced a bug fixing, described in the Issue #2568 [shadow not saved]: *"the shadow correctly work during the drawing of an UML diagram but, if the diagram is saved, closed, and re-opened, the shadow is no more present in it"*.

Other refactoring operations often inducing bug fixes are *Inline Temp*, *Replace Method With Method Object*, and *Extract Method*. Among them, the last is the most diffused, applied when a long method contains one (or more) code fragments implementing a precise responsibility. The code fragment is moved in a new method having a name explaining its purpose. The goal is clearly to decompose a long method into smaller, easier to comprehend, methods. This refactoring induces an error in 79 out of the 374 refactored classes (21%). An *Extract Method* refactoring inducing a fix is the one performed between the versions 0.14 and 0.18.1 of ArgoUML in the class *ActionAddDiagram*, parent of all the classes implementing actions adding diagrams in ArgoUML. In particular, from the method *actionPerformed(ActionEvent e)*, the new method *findNamespace()* has been extracted. A wrong swap of order between two instructions during the performed *Extract Method* has introduced the following bug (Issue #2287): *"When you create a new diagram, the new diagram node is added to the explorer tree, but it is not selected"*.

We summarize the likelihood of refactorings to induce fixes by analyzing how percentages of likely fault prone refactorings are distributed: the first quartile is 6%, the median is 13%, and the third quartile is 18%. Figure 2 classifies the analyzed refactorings in *Not Harmful* (Prop. $<$ 6%), *Potentially Harmful* (6% $\leq$ Prop. $<$ 18%), or *Harmful* (Prop. $\geq$ 18%)

Summary for RQ$_2$: while, in general, the median percentage of fault-prone refactored classes is relatively low (i.e., 13%), there are some specific kinds of refactorings—e.g., Pull-up method or Inline Temp—that are very likely to induce bug fixes.

C. RQ$_3$: Are refactorings more likely to induce bug fixes in source or target components?

Table VII reports, for each refactoring operation involving more than one class, (i) the total number of bugs found in classes related to the refactoring (column #Total), (ii) the number of bugs found in the source classes (column #Source), and (iii) the number of bugs found in the target classes (column #Target). Note that for some refactoring operations, the sum of #Source and #Target is not equals to #Total since source and target classes coincide.

Some of our findings are easy to explain. For example, *Inline Class* refactoring never induces bug fixes on the source class. The reason is that with this type of refactoring, the entire source class is moved inside the target class, and thus the source class no longer exists after the refactoring operation. With *Inline Method*, i.e., the body of a method m is put inside another method and m is deleted, the source and target classes always coincide and, thus, are trivially, equally impacted.

More interesting are, for example, the operations of *Move Field* and *Move Method*. For both these refactorings most of the bug fixes are induced in the target class (69% for the *Move Field* and 60% for the *Move Method*). We manually analyzed some of these cases to understand the reasons behind these results. As for the *Move Field*, this is generally due to a wrong initialization of the moved variable in the new class (and thus, in the target class). As for the *Move Method*, we observed some errors due to the overriding of a method inherited by the target class from its superclass.

Also interesting are the case of *Replace Method With Method Object* and *Pull Up Method* refactoring. As for the first, it induces a bug fixing in the source class in 80% of cases. This refactoring moves some fields of a long method (implemented in the source class) inside a new object (the

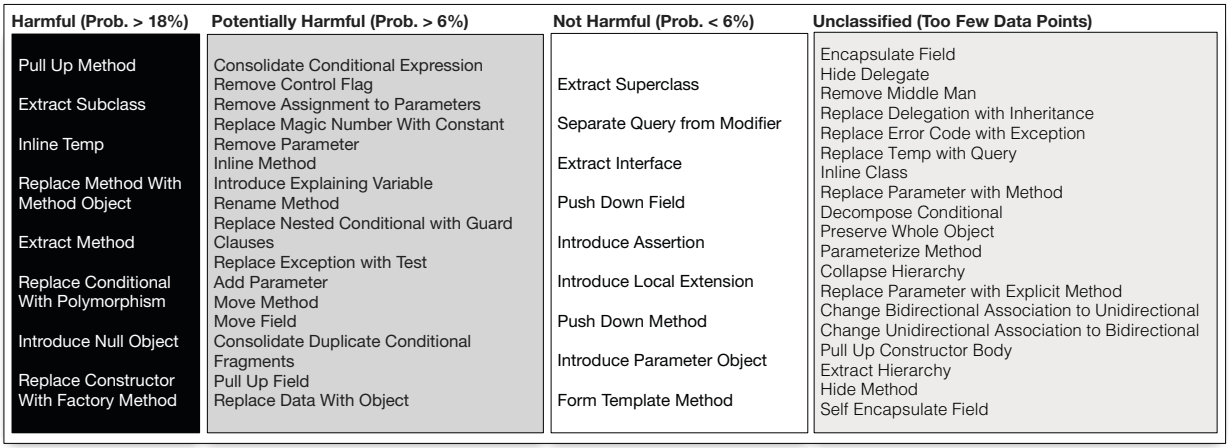

Fig. 2. Summary of fault-proneness for different kinds of refactoring operations

TABLE VII
BUGS FOUND IN SOURCE AND TARGET CLASSES

Operation	Ant Apache			ArgoUML			Xerces			Total		
	#Total	#Source	#Target	#Total	#Source	#Target	#Total	#Source	#Target	#Total	#Source	#Target
Change Bidirectional Association to Unidirectional	0	0	0	5	0	5	1	0	1	6	0	6
Change Unidirectional Association to Bidirectional	0	0	0	0	0	0	9	3	6	9	3	6
Collapse Hierarchy	0	0	0	0	0	0	5	0	5	5	0	5
Extract Hierarchy	0	0	0	6	6	0	0	0	0	6	6	0
Extract Interface	0	0	0	12	0	12	6	0	6	18	0	18
Extract Subclass	0	0	0	11	6	5	3	3	0	14	9	5
Extract Superclass	0	0	0	5	5	0	0	0	0	5	5	0
Form Template Method	0	0	0	0	0	0	0	0	0	0	0	0
Inline Class	14	0	14	0	0	0	0	0	0	14	0	14
Inline Method	16	0	16	57	0	57	125	0	125	198	198	198
Move Field	125	6	119	386	146	240	121	47	74	632	199	433
Move Method	83	6	77	613	251	362	212	108	104	908	365	543
Pull Up Constructor Body	0	0	0	0	0	0	0	0	0	0	0	0
Pull Up Field	0	0	0	0	0	0	4	0	4	4	0	4
Pull Up Method	0	0	0	0	0	0	30	30	0	30	30	0
Push Down Field	0	0	0	0	0	0	3	0	3	3	0	3
Push Down Method	0	0	0	0	0	0	0	0	0	0	0	0
Remove Middle Man	0	0	0	0	0	0	0	0	0	0	0	0
Replace Conditional With Polymorphism	0	0	0	11	6	5	0	0	0	11	6	5
Replace Data With Object	1	1	0	17	0	17	4	2	2	22	3	19
Replace Delegation With Inheritance	0	0	0	0	0	0	0	0	0	0	0	0
Replace Method With Method Object	0	0	0	870	615	255	490	479	11	1,360	1,094	266

target class, generally represented by an entity class). Thus, the target class involved in this refactoring is generally very simple and, for this reason, rarely subject to bug-fix inducing changes.

Finally, concerning the *Pull Up Method*, Table VII shows how the bug-fixes induced by this refactoring only affect the source classes. Manual inspection of particular cases highlights as often the method pulled-up in the superclass is slightly modified to meet the needs of all the subclasses, resulting sometimes in wrong behavior in the subclasses.

As for the statistical analysis, on the Ant system, statistical tests do not show significant differences. On ArgoUML, they show that *Move Field* and *Move Method* have a significantly higher chance to induce bug fixes in target classes than in source classes if compared with *Replace Method with Method Object* (OR=8.94 and 6.11 respectively). Similar results, but with higher ORs (67.65 and 41.59 respectively) were found for Xerces.

Summary for RQ$_3$: refactorings like *Move Field* and *Move Method* are more prone to inducing errors in the target classes, while *Replace Method With Method Object* and *Pull Up Method* mostly induce bugs in the source classes. For other kinds of refactoring, no particular observations can be done.

V. THREATS TO VALIDITY

This section describes the threats that can affect the validity our study.

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is probably the most important kind of threat for this study, and is related to

- *imprecision in the identification of refactorings:* this was mitigated through a manual validation;
- *missing or wrong links between bug tracking systems and versioning systems* [15]: although not much can be done for missing links, we verified that links between commit notes and issues are correct;

- *imprecision in issue classification made by issue-tracking systems* [16]: at least the three systems we consider use an explicit classification of bugs on issue tracking systems, distinguishing them from other issues;
- *approximations due to identifying bug-inducing changes using the SZZ algorithm* [13]: at least we used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of bug-inducing changes;
- *refactorings identified at release level*: in order to precisely link bug-fix introducing changes, ideally one should perform refactoring detection on each change occurred in the system. However, in this first study we performed the analysis at release level for two reasons (i) practical reason (due to the huge number of system snapshots to be analyzed), and, above all, (ii) the need for manually validating the detected refactorings, which would have been unfeasible if performed at change-level rather than at release-level. It has to be considered, however, that all the three systems analyzed tend to issue releases quite frequently (as evident from the number of analyzed releases), therefore the analysis at release level is not excessively coarse-grained.

Threats to *internal validity* concern external factors we did not consider that could affect the variables being investigated. Such threats are related to lack of causation between refactorings and bug fixes. At least, we integrated the quantitative with qualitative ones, plus examples we found by browsing source code and data from versioning/issue tracking systems.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalization of results. We analyzed three different systems, two belonging to the same family (Apache). However, further systems should be analyzed to confirm or contradict our conclusions. Also, as mentioned above we rely on refactorings identified by *RefFinder*, hence the study excludes refactoring that such a tool does not identify. For example, *extract class* refactoring is not handled, but it is identified as multiple *move method*.

VI. Related Work

In the refactoring field, most of the effort has been devoted to the definition of automatic and semi-automatic approaches supporting refactoring operations (see [17], [18] for a complete survey on the more recent approaches). However, our paper is mostly related to work (i) analyzing the relation between refactorings and software defects and (ii) methods and tools to detect refactoring operations occurring between subsequent versions of a software system. In the following, we discuss the related literature.

A. Refactoring and Software Defects

To the best of our knowledge, only two works have dealt with analysis related to refactorings and software defects. The first is the one by Weissgerber and Diehl [9] where the authors analyze if refactoring operations are less error-prone than other changes. They browsed the history of three open source systems and, for each day, stored (i) the changes applied in the repository, (ii) the number of refactorings applied, and (iii) the number of bugs opened. Then, the authors analyzed if days in the projects with a high number of refactorings performed are followed by days with a high number of bugs opened. Their results showed that in general there is no clear correlation between the number of refactoring operations and the number of bugs opened in the following days. However, there are phases where there was a strong increase of the number of bugs opened in consequence of a high refactoring activity on the system.

Our work has several differences with the work by Weissgerber and Diehl [9]. First of all, Weissgerber and Diehl [9] automatically identify only 8 kinds of refactoring operations in the releases of the analyzed software projects. In addition, the identified refactorings are not manually validated. In this paper, we automatically identify (using Ref-Finder) 52 different kinds of refactoring operations and manually validate each of them. Another important difference is related to the granularity level of the performed analysis. In particular, while Weissgerber and Diehl [9] perform a coarse-grained analysis by relating the number of refactoring operations performed with the number of bugs opened in the following days, we analyzed to what extent each of the 52 kinds of detected refactoring operations induces a fault-fixing in the system using the SZZ algorithm.

A second study related to refactorings and software defects is the one presented by Ratzinger *et al.* [8]. The authors of this paper analyze if (i) refactoring history information is useful to support defect prediction and (ii) refactoring activities reduce the probability of software defects. They show that refactorings and defects have an inverse correlation: the number of software defects decreases, if the number of refactorings increased in the preceding time period. Differently from our work, Ratzinger *et al.* do not distinguish among different kinds of refactoring operations. Moreover, their detection process is not performed by source code analysis of subsequent system's version, but through the analysis of the text contained in change logs.

B. Detecting Refactoring Operations

Different approaches have been defined to detect refactoring operations between different versions of a software system. In this section we only discuss some of them while a complete discussion has been presented by Prete *et al.* [11].

Demeyer *et al.* [19] present an approach based on changes in size metrics (e.g., lines of code, number of methods) to identify performed refactoring operations (e.g., the reduction of the number of methods in a class might be a consequence of a move method refactoring). Only 7 types of refactoring

operations are detected in this work. Godfrey and Zou [20] present an approach to detect merge, split, and rename refactorings, while Van Rysselberghe and Demeyer [21] use clone detection to identify move method refactoring operations.

Dig *et al.* [22] present RefactoringCrawler, an Eclipse plug-in using a two-step process to detect refactoring operations. In the first step, a fast syntactic analysis is applied to detect refactoring candidates, while a more expensive semantic analysis is applied to refine the results. RefactoringCrawler supports the detection of 16 different refactoring operations.

Weissgerber and Diehl [23] show a technique to detect ten different kinds of refactorings, while Xing and Stroulia [24] present an approach to identify up to 32 different kinds of refactorings. Untill now, no one has provided a broader support to the detection of refactoring operation than that offered by Ref-Finder [11]. Such a tool is able to detect 63 kinds of refactorings. This is the reason why we decided to use Ref-Finder to automatically detect refactorings in our study.

VII. Conclusion and Work-in-Progress

In this paper we reported an empirical analysis aimed at at investigating the extent to which refactoring activities induce bug fixes in software systems. Indeed, while refactoring activities would potentially produce benefits, as well as any other change occurring in a software system they might also be harmful and introduce bugs.

We used Ref-Finder to automatically detect 15,008 refactoring operations (of 52 different types) on 63 releases of three Java software systems. Then, we manually validated all the identified refactoring operations in order to eliminate false positives. Among the operations identified by Ref-Finder, 12,922 were actual refactoring operations. Then, we used the SZZ algorithm [12], [13] to determine whether refactorings induced bug fixes.

The achieved results showed that the percentage of faults likely induced by refactorings is relatively low (i.e., 15%). However, there are some specific kinds of refactorings that are very likely to induce bug fixes, such as *Pull Up Method* and *Extract Subclass*, where the percentage of fixes likely induced by such refactorings is around 40%. Such a result suggests more accurate code inspection or testing activities when such specific refactorings are performed.

Future work will be devoted to replicate the study with other software systems. As it always happens with empirical studies, this is the only way to corroborate our findings.

References

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press, London, 1985.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 1999.

[3] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering*. Sorrento, Italy, IEEE CS Press, 1994, pp. 279–287.

[4] W. Harrison, "An entropy-based measure of software complexity," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 1025–1029, 1992.

[5] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. Vancouver, Canada, ACM Press, 2009, pp. 78–88.

[6] M. Fowler, "Refactoring catalog." [Online]. Available: http://refactoring.com

[7] G. Canfora, L. Cerulo, M. Di Penta, and F. Pacilio, "An exploratory study of factors influencing change entropy," in *Proceedings of the 18th International Conference on Program Comprehension*. Braga, Portugal, IEEE Computer Society, 2010, pp. 134–143.

[8] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. Leipzig, Germany, ACM Press, 2008, pp. 35–38.

[9] P. Weissgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*. Shanghai, China, ACM Press, 2006, pp. 112–118.

[10] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Dubrovnik, Croatia, ACM Press, 2007, pp. 185–194.

[11] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*. Timisoara, Romania, IEEE CS Press, 2010, pp. 1–10.

[12] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*. Saint Louis, Missouri, USA, ACM Press, 2005.

[13] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[14] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Amsterdam, The Netherlands, ACM Press, 2009, pp. 121–130.

[16] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Centre for Advanced Studies on Collaborative Research*. Richmond Hill, Ontario, Canada, IBM Press, 2008, p. 23.

[17] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[18] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, pp. 397–414, March 2011.

[19] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *SIGPLAN Notes*, vol. 35, no. 10, pp. 166–177, Oct. 2000.

[20] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[21] F. Van Rysselberghe and S. Demeyer, "Reconstruction of successful software evolution using clone detection," *Proceedings of the International Workshop on Principles of Software Evolution*. Helsinki, Finland, IEEE Press, pp. 126–130, 2003.

[22] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming* Nantes, France, Springer-Verlag, 2006, pp. 404–428.

[23] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Tokyo, Japan, IEEE Computer Society, 2006, pp. 231–240.

[24] Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Benevento, Italy, IEEE CS Press, 2006, pp. 263–274.