An Exploratory Study on the Relationship between Changes and Refactoring

Fabio Palomba*, Andy Zaidman*, Rocco Oliveto‡, Andrea De Lucia†
*Delft University of Technology, The Netherlands - †University of Salerno, Italy - ‡University of Molise, Italy
f.palomba@tudelft.nl, a.e.zaidman@tudelft.nl, rocco.oliveto@unimol.it, adelucia@unisa.it

***Abstract*—Refactoring aims at improving the internal structure of a software system without changing its external behavior. Previous studies empirically assessed, on the one hand, the benefits of refactoring in terms of code quality and developers' productivity, and on the other hand, the underlying reasons that push programmers to apply refactoring. Results achieved in the latter investigations indicate that besides personal motivation such as the responsibility concerned with code authorship, refactoring is mainly performed as a consequence of changes in the requirements rather than driven by software quality. However, these findings have been derived by surveying developers, and therefore no software repository study has been carried out to corroborate the achieved findings. To bridge this gap, we provide a quantitative investigation on the relationship between different types of code changes (*i.e.*, *Fault Repairing Modification*, *Feature Introduction Modification*, and *General Maintenance Modification*) and 28 different refactoring types coming from 3 open source projects. Results showed that developers tend to apply a higher number of refactoring operations aimed at improving maintainability and comprehensibility of the source code when fixing bugs. Instead, when new features are implemented, more complex refactoring operations are performed to improve code cohesion. Most of the times, the underlying reasons behind the application of such refactoring operations are represented by the presence of duplicate code or previously introduced self-admitted technical debts.**

***Index Terms*—Refactoring; Code Changes; Empirical Studies**

I. Introduction

Refactoring is *"the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"* [1]. The value of refactoring has been widely demonstrated in the past, since it improves the internal structure of the source code leading to several positive effects, such adaptability, maintainability, understandability [2], reusability, and testability [3], [4], but also developers' productivity [5]. Moreover, the higher the number of refactoring operations performed by developers the higher the benefits for software maintainability [6].

These empirical studies have motivated researchers in spending effort for devising techniques able to discover areas of source code needing refactoring [7], [8], [9], as well as methods for the identification of refactoring opportunities [10], [11], [12], [13]. Despite this effort, developers tend to not refactor source code as they should, and they generally do not use any automated tool to improve the quality of a system [14]. With the aim of supporting developers in such an activity, in the recent past some studies have further investigated how programmers apply refactoring [15], and what are the conditions pushing them to apply refactoring operations [16].

Such studies showed that in most cases refactoring is not recognized as a behavior-preserving operation [15] and, thus, developers perform refactoring (i) only when strictly needed to implement new features (*e.g.*, when the source code is poorly readable [15]), (ii) because of the responsibility concerned with code authorship [17], or (iii) to achieve recognitions from others [17].

More recently, Bavota *et al.* [18] analyzed to what extent refactoring operations are performed on classes having a low metric profile or affected by code smells. They found that refactoring operations do not target classes exhibiting low cohesion and/or high coupling, and that only 40% of the times refactoring operations have been performed on classes affected by a design flaw. These results have also been confirmed by Silva *et al.* [16], which surveyed the `Github` contributors of 124 software projects, finding that refactoring is mainly driven by changes in the requirements rather than by the presence of quality problems in source code (*e.g.*, code smells).

While Silva *et al.* [16] explored the problem from a developers' perspective, there are no studies that systematically investigate software repositories to understand whether specific types of changes drive refactoring operations. To bridge this gap, in this paper we empirically verify the relationship between the types of changes coming from the taxonomy provided by Hassan [19] (*i.e.*, *Fault Repairing Modification*, *Feature Introduction Modification*, and *General Maintenance Modification*), and the application of refactoring operations. The study has been conducted on a dataset composed of 12,922 operations related to 28 different refactoring types applied over the change history of three open source systems, *i.e.*, APACHE ANT, ARGOUML, and APACHE XERCES.

The results of the study firstly indicate that classes having a higher rate of fault repairing modifications have higher chance of being subject to refactoring operations aimed at simplifying the source code (by improving its comprehensibility) and improving its maintainability (*e.g.*, through a *move field* refactoring). A deeper investigation into the reasons why developers apply such refactoring operations during bug fixing activities revealed that in 74% of the cases the main reason for developers to re-organize the code is the presence of duplicated code [1]. At the same time, we observed that in 96% of the cases the overall readability of refactored classes is improved by 48% (as indicated by the Buse and Weimer readability metric [20]). Furthermore, feature introduction changes have a higher likelihood of entailing refactoring opeartions aimed at improving code cohesion or adherence to the Object-Oriented programming principles (*e.g.*, through an *extract method* refactoring). In this case, we observed that 46% of the refactored classes have been affected by a self-admitted technical debt in their previous versions. Therefore, most of the times refactoring can be seen as a form of compensation of pre-existing debts. Finally, general maintenance modifications lead to improve the readability of the code (*e.g.*, by applying a *rename method* refactoring). As a result, the overall readability of the refactored classes increase of 30%.

Structure of the Paper. Section II describes the design of our empirical study, while Section III reports and discusses the obtained results. Section IV analyzes and discusses the threats that could affect the validity of our study. After a discussion of the related literature (Section V), Section VI concludes the paper.

II. Empirical Study Definition and Design

The *goal* of the empirical study is to analyze refactoring operations applied by developers during the evolution history of a software system. The *purpose* is understanding whether different types of changes provide indications on which code components are more/less likely of being refactored.

The *context* of the study consists of 63 releases of three software projects with different size and scope, *i.e.,* Apache Ant[1], ArgoUML[2] and Apache Xerces-J[3]. The characteristics of the object systems are reported in Table I.

It is worth noting that we focus our attention on a relative small number of software systems because, as detailed in Section II-A, we relied on a publicly available dataset composed of 12,922 operations (manually validated) related to 28 different refactoring types identified in each of the considered releases [18]. Note that while other datasets are available [21], to the best of our knowledge the one built by Bavota *et al.* is the largest one in terms of refactoring operations (12,922 vs 7,872 reported by Kadar *et al.*).

A. Research Questions and Data Extraction

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

To what extent are refactoring operations performed on classes subject to a fault repairing, feature introduction, and general maintenance modification?

To answer our research question, we firstly needed to identify which types of refactoring operations occur over the history of the considered software systems. The dataset of refactoring operations we relied on [18] reports a set of 12,922 refactoring operations applied over 63 releases of the three

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

TABLE I: Characteristics of the object systems.

Project	Period	Releases Analyzed	#Releases	Classes	KLOC
Ant	Jan 2000-Dec 2010	1.2-1.8.2	17	87-1,191	8-255
ArgoUML	Oct 2002-Dec 2011	0.12-0.34	13	777-1,519	362-918
Xerces-J	Nov 1999-Nov 2010	1.0.4-2.9.1	33	181-776	56-179
Overall	-	-	**63**	-	-

object systems. Specifically, the dataset is composed of a set of triples (rel_j, ref_k, C), where rel_j indicates the release ID, ref_k the type of refactoring that occurred, and C is the set of refactored classes. Therefore, the dataset reports all the information needed to apply our analyses. Table II shows the number of refactoring operations (together with the number of different types of refactoring operations) identified on the three systems after the manual validation process.

To extract the different types of changes involving classes affected by refactoring operations across two consecutive releases of the analyzed software systems, we mined the logs of their versioning systems. Specifically, we discriminate three different types of changes, following the taxonomy proposed by Hassan [19]:

- **Fault Repairing Modification (FR)**, which represents the set of changes applied to fix a fault. Such changes are usually specified by developers in the commit message through the indication of the ID of the fault the commit repairs (*e.g.*, "Issue #42 fixed").
- **Feature Introduction Modification (FI)**, representing the set of changes adding or enhancing a given feature. It is possible to discriminate such changes looking for keywords as "added" or "updated" in the commit message.
- **General Maintenance Modification (GM)**, namely the set of changes not related to the update of a specific feature. For example, the modification of the indentation of the source code can be considered as a GM.

We automatically classified each commit by applying the lexical technique proposed by Mockus *et al.* [22], that is able to assign a category of change based on the analysis of the commit message.

B. Study Variables and Analysis Method

The *dependent variables* of our study are the different types of refactoring operations performed over all the releases of the software projects we considered. The *independent variables* are instead the different types of changes we related to the observed refactoring operations. For each system in our dataset and for each type of refactoring applied to it, we built logistic regression models[4] that relate a dichotomous dependent variable with independent variables characterized by the change factors. In other words, given the set of independent variables, we are interested in the prediction of the probability p that the dependent variable is 1 (*i.e.*, the refactoring occurs) rather than 0 (*i.e.*, the refactoring does not occurs).

Logistic regression models [23] relate dichotomous dependent variables with one or more independent variables as follows:

[4]Using the R statistical software: http://www.r-project.org/

TABLE II: Summary of the refactoring operations analyzed.

Project	#Refactorings	Distinct types of refactorings
Apache Ant	1,469	31
ArgoUML	3,532	43
Xerces-J	7,921	43
Overall	**12,922**	**52**

$$\pi(X_1, X_2, \dots, X_n) = \frac{e^{C_0 + C_1 \cdot X_1 + \dots + C_n \cdot X_n}}{1 + e^{C_0 + C_1 \cdot X_1 + \dots + C_n \cdot X_n}} \quad (1)$$

where X_i are the independent variables characterizing an event, and C_i the coefficients (estimates) of the logistic regression model. It is worth noting that, to avoid the definition of unreliable logistic regression models, we choose to apply the logistic regression model only if a particular type of refactoring (*e.g.*, *Move Field* refactoring) has been performed on a system at least ten times. In particular, we built three different logistic models:

1) **FR Model**: This model considers fault repairing modifications as independent variables, while the application of a specific refactoring (*e.g.*, *add parameter*) as dependent variable.
2) **FI Model**: The second model considers feature introduction modifications as independent variables, while the application of a specific refactoring (*e.g.*, *add parameter*) as dependent variable.
3) **GM Model**: The last model considers general maintenance modifications as independent variables, while the application of a specific refactoring (*e.g.*, *add parameter*) as dependent variable.

For each considered model we then analyze if each independent variable is significantly correlated with the dependent variable (we set the significance level $\alpha = 5\%$), and we quantify the correlation between the variables using the Odds Ratio (OR) [24] which, for a logistic regression model, is given by e^{C_i}. In our case, Odd Ratios indicate the increase in likelihood of a refactoring increase/decrease as a consequence of a one-unit increase of the independent variable. For example, if we found that *Feature Introduction Modification* has an OR of 1.10 with *extract method* refactoring, this means that each one-unit increase of the feature introduction modification made on a class lead to a 10% higher chance for the class of being involved in an *extract method* refactoring.

Besides the analysis made to understand the relationship between changes and refactoring from a quantitative perspective, we performed a complementary qualitative investigation into the source code of the classes refactored by developers, with the aim to understand the underlying reasons behind the application of a given refactoring. In particular, we manually analyzed the commit messages and the source code involving the artifacts refactored during the history of the considered systems with the purpose of analyzing whether classes subject to refactoring have particular characteristics making them more prone to be re-organized by developers. More details are reported along with the discussion of the results.

III. Analysis of the Results

Table III reports the ORs of the fault repairing modifications, feature introduction modifications, and general maintenance modifications, respectively, for the 28 different types of refactoring operations considered in the study. Statistically significant ORs are highlighted in bold face. In the following, we discuss the results of the study by considering each model independently.

FR Model. As it is possible to observe from Table III, 80% of the statistically significant ORs are higher than one. From a practical point of view, this means that classes having a higher rate of fault repairing modifications have a higher chance of being refactored than classes not involved in bug fixing activities. Likely, this is due to the fact that developers want to simplify the structure of a class in order to make it more understandable before applying delicate corrective maintenance operations. This claim is supported by the fact that most of the refactoring types exhibiting high ORs deal with simplifying the source code and improving its comprehensibility. It is the case for *add parameter*, *consolidate duplicate conditional fragments*, *move field*, *remove assignment to parameters*, *replace magic number with constant*, and *replace nested cond guard clauses*. Note that we additionally verified whether the number of refactoring operations having higher ORs (*i.e.,* in the case of fault repairing modifications, the ones mentioned above) was statistically higher than the number of the other refactoring operations. To this aim, we exploited the Mann-Whitney U test [25] comparing the distributions of refactoring operations among the three subject systems. As a result, we observed that the refactoring types having higher ORs have been actually applied a statistically higher number of times with respect to the other refactoring types ($\alpha < 0.01$ in all the cases).

Particularly interesting is the case of the *add parameter* refactoring which has OR=13.18 for Apache Ant, OR=23.10 for ArgoUML, and OR=10.95 for Xerces. By analyzing more in depth these cases, we found that often refactoring is an absolute need for developers to effectively perform bug fixing activities: for instance, a developer of ArgoUML, after having refactored the source code and applied the bug fix, committed the new version of the class reporting this commit message:

> *"Fixed bug #221148. I needed to add parameters and comments in the class, because it was totally horrible and impossible to fix!"*

This result is even more interesting when we consider the more general relation between refactoring and bugs. Indeed, as Bavota *et al.* have shown [26], refactoring operations that are applied manually by developers could induce bug fixing activities in the source code. In contrast, we show that bug fixing activities make developers more prone in their application of specific refactoring operations. So, the relation seems to be bidirectional and our findings highlight the need

TABLE III: ORs achieved by logistic regression models built using fault repairing, feature introduction, and general maintenance modifications (statistically significant ORs are in bold face).

Refactoring	System	FR Modifications	FI Modifications	GM Modifications
add parameter	Apache Ant	**13.18**	**8.17**	4.81
add parameter	Argo UML	**23.10**	**3.15**	**2.91**
add parameter	Xerces	**10.95**	**22.87**	**1.11**
consolidate cond expression	Apache Ant	0.53	1.11	**1.74**
consolidate cond expression	Argo UML	0.59	1.55	1.55
consolidate cond expression	Xerces	0.78	**1.13**	**2.14**
consolidate duplicate cond fragments	Apache Ant	**0.61**	0.98	**1.31**
consolidate duplicate cond fragments	Argo UML	**0.74**	0.70	0.44
consolidate duplicate cond fragments	Xerces	0.63	0.79	**1.57**
extract method	Apache Ant	**1.02**	**5.18**	0.71
extract method	Argo UML	0.89	**9.25**	0.78
extract method	Xerces	**1.07**	**3.11**	**0.80**
extract superclass	Argo UML	**5.81**	0.83	0.68
form template method	Argo UML	0.83	**2.46**	3.49
inline method	Apache Ant	0.75	1.89	**0.66**
inline method	Argo UML	0.69	**1.98**	**0.70**
inline method	Xerces	**1.71**	**1.65**	0.81
inline temp	Apache Ant	**1.52**	**0.81**	**0.86**
inline temp	Argo UML	1.01	**0.87**	0.69
inline temp	Xerces	**0.79**	1.76	**0.82**
introduce assertion	Argo UML	1.01	1.12	**0.99**
introduce explaining variable	Apache Ant	0.88	**4.18**	**5.00**
introduce explaining variable	Argo UML	**1.06**	**0.85**	**18.23**
introduce explaining variable	Xerces	**1.02**	1.01	**2.68**
introduce null object	Argo UML	0.99	0.74	0.89
introduce parameter object	Xerces	2.76	**1.16**	0.86
move field	Apache Ant	**7.98**	**1.08**	**3.41**
move field	Argo UML	**5.14**	1.19	**2.87**
move field	Xerces	**8.34**	**1.76**	2.04
move method	Apache Ant	5.86	**1.17**	**1.02**
move method	Argo UML	**3.91**	1.12	**4.41**
move method	Xerces	**2.76**	0.99	2.15
pull up field	Xerces	0.91	0.88	0.52
pull up method	Xerces	1.07	0.90	0.78
push down field	Xerces	1.86	**1.24**	**0.92**
push down method	Xerces	0.80	**2.98**	**0.55**
remove assignment to parameters	Apache Ant	**2.12**	0.69	1.11
remove assignment to parameters	Argo UML	**1.23**	0.96	**2.71**
remove assignment to parameters	Xerces	**0.88**	0.78	0.70
remove control flag	Apache Ant	**2.13**	**0.98**	0.61
remove control flag	Argo UML	4.13	**0.91**	0.82
remove control flag	Xerces	**1.19**	0.71	0.33
remove parameter	Apache Ant	1.02	0.86	**1.22**
remove parameter	Argo UML	0.66	**0.88**	2.88
remove parameter	Xerces	0.87	0.91	**3.61**
rename method	Apache Ant	1.21	0.93	**14.11**
rename method	Argo UML	**1.12**	**4.87**	**1.58**
rename method	Xerces	1.75	**1.07**	**3.73**
replace data with object	Argo UML	1.39	**2.98**	3.81
replace data with object	Xerces	0.91	**8.16**	2.04
replace exception with test	Xerces	0.80	0.92	**1.09**
replace magic number with constant	Apache Ant	**1.18**	0.78	**1.02**
replace magic number with constant	Argo UML	**13.72**	**0.97**	0.59
replace magic number with constant	Xerces	**0.66**	**1.01**	2.75
replace method with method object	Apache Ant	0.94	**4.09**	5.79
replace method with method object	Argo UML	**1.32**	**1.98**	**1.35**
replace method with method object	Xerces	1.53	2.71	**12.81**
replace nested cond guard clauses	Apache Ant	0.71	0.88	0.16
replace nested cond guard clauses	Argo UML	**1.09**	**1.56**	0.45
replace nested cond guard clauses	Xerces	**0.76**	0.99	0.22
separate query from modifier	Xerces	0.83	0.80	**1.55**

to further investigate the interaction between refactoring and bugs.

For this reason, we have performed a deep analysis of the change history of the subject systems by manually inspecting the commit messages and the source code related to commits having as goal the fixing of bugs (as indicated by the commits' classification automatically done using the approach by Mockus [22]). From this additional analysis, we learned that in 74% of the cases the commits involved in refactoring operations contain source code affected by duplicated code [1]. Developers refactored these affected parts by applying operations aimed at improving the comprehensibility and/or the maintainability of the source code before fixing a bug. This finding is quite unexpected if we consider that all the refactoring operations having higher ORs are not specifically targeted at removing code clones [1]. However, most of the operations performed by developers (*e.g., consolidate duplicate conditional fragments*) tend to re-unify the source code

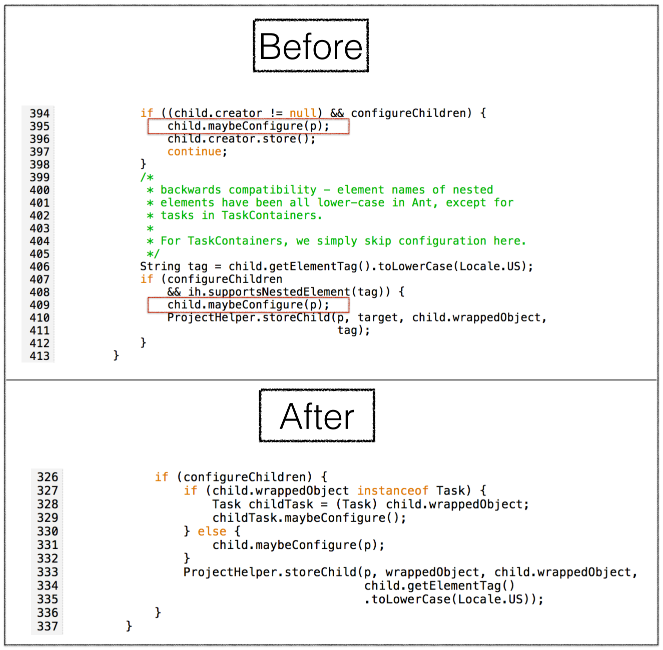


Fig. 1: Method `maybeConfigure` of the Apache Ant project before and after the refactoring operations applied to fix a bug.

```java
394             if ((child.creator != null) && configureChildren) {
395                 child.maybeConfigure(p);
396                 child.creator.store();
397                 continue;
398             }
399             /*
400              * backwards compatibility - element names of nested
401              * elements have been all lower-case in Ant, except for
402              * tasks in TaskContainers.
403              *
404              * For TaskContainers, we simply skip configuration here.
405              */
406             String tag = child.getElementTag().toLowerCase(Locale.US);
407             if (configureChildren
408                 && ih.supportsNestedElement(tag)) {
409                 child.maybeConfigure(p);
410                 ProjectHelper.storeChild(p, target, child.wrappedObject,
411                                          tag);
412             }
413         }
```

```java
326             if (configureChildren) {
327                 if (child.wrappedObject instanceof Task) {
328                     Task childTask = (Task) child.wrappedObject;
329                     childTask.maybeConfigure();
330                 } else {
331                     child.maybeConfigure(p);
332                 }
333                 ProjectHelper.storeChild(p, wrappedObject, child.wrappedObject,
334                                          child.getElementTag()
335                                          .toLowerCase(Locale.US));
336             }
337         }
```

by removing redundant code. A clear example is represented by the class `RuntimeConfigurable` of the Apache Ant system, where the `maybeConfigure` method is in charge of configuring the proper build properties for a new Java project. The upper part of Fig. 1 depicts a snippet of code (from line #385 to line #413 of the class) referring to the investigated method. The code snippet shows that the method can call itself (red lines in Figure 1) in two different `if` statements (lines #394 and #413). In version 1.6.1 of the system, the method was affected by a known bug causing a double configuration of the project if the input file contains sub-tasks.[5] When solving this bug, the developers first applied a *consolidate duplicate conditional fragments* refactoring aimed at condensing the two conditional statements leading to two different calls of the `maybeConfigure` method into a single one (see the lower part of Fig. 1). At the same time, the bug was fixed by applying an *add parameter* refactoring in order to pass the method a boolean variable named `configureChildren` able to control whether the input project needs or does not need the configuration of its sub-projects.

When refactoring is not applied to remove redundant code, developers perform modifications on fields and local variables aimed at improving their location or their names. So, all in all, we observed that developers performing bug fixing activities apply refactoring operations for two possible reasons: (i) improving the general maintainability of the system, or (ii) improving the comprehensibility of source code before fixing a fault. This result is in line with previous findings by Du Bois *et al.*, who have originally shown that *"refactoring to understand"* is one of the main activities performed by developers when conducting maintenance operations [27]. To further corroborate the latter statement, we also verified whether the source code refactored during bug fixing showed an improvement in its overall readability. To this aim, we exploited the metric proposed by Buse and Weimer [20]. This metric combines a set of low-level code features (*e.g.,* identifier length, number of loops, *etc.*) and has been shown to be 80% effective in predicting developers' readability judgments. We used the original implementation provided by the authors of the metric.[6] In particular, given a code file, the readability metric takes values between 0 (lowest readability) and 1 (maximum readability). From this analysis, we obtained that in 96% of the cases the refactored classes obtained an average improvement of 48% of the readability score. Thus, we can confirm that the refactoring operations made during bug fixing activities have a beneficial effect on program comprehensibility, other than the maintainability of a software system. On the other hand, we observed that code clones (*i.e.,* one of the most popular code smells [1]) represent the main reason why refactoring is applied during bug fixing activities. Our qualitative findings confirm the results reported by Silva *et al.* [16], where the authors found that the presence of code clones represent a notable motivation for refactoring the code.

Observation 1. During bug fixing activities, developers improves comprehensibility and maintainability of the source code. The main reason pushing developers to refactor source code is the presence of duplicated code. At the same time, we also found that in 96% of the cases the readability of the source code refactored during bug fixing operations showed an improvement of 48%.

FI Model. The results for the model involving the feature introduction modifications are reported in Table III. Also in this case, a large part of statistically significant ORs are higher than one (*i.e.*, 78% of the cases). Moreover, for *add parameter*, *extract method*, *replace data with object*, and *replace method with method object* refactoring operations, such ORs are consistently higher than one, indicating that all of them are closely related to the introduction of new features in a software system. Indeed, the number of times these refactoring operations have been applied is statistically higher than the one of all the other refactoring operations. The result is somehow expected, since developers implementing new features need to re-organize specific parts of the system in order to place the new requirements in the right classes. Therefore, refactoring operations as *extract method* or *replace data with object* are perfectly inline with our conjecture. A clear example occurred in the Apache Ant project, where a developer implementing the option `-noclasspath`[7] had to modify the source code

[5]https://bz.apache.org/bugzilla/show_bug.cgi?id=9900

[6]Available at http://tinyurl.com/kzw43n6

[7]The option used to run ant without using the classpath of a project.

of the class `org.apache.tools.ant.Task`. To this aim, she applied an *extract method* refactoring in order to extract from the method `handleInput` (*i.e.,* the method in charge of analyzing the input of the project) the part related to the management of the default input provided by the user. The extracted part has then been placed in a new method named `defaultInput`. As a direct consequence, the overall cohesion of the class was improved (*i.e.,* the LCOM—Lack of Cohesion of Methods [28]—decreases from 6 to 2). Interestingly, before the refactoring, the method `handleInput` was associated with the following comment:

```
    /* It can produce errors in
  older versions. Need fix (sooner
  or later). */
```

It seems that in an older version of the system the developers consciously left a possible issue into the system with the aim of speeding up the release process. Thus, they introduced a *self-admitted technical debt* [29], that was subsequently payed off during the implementation of a new feature involving the method `handleInput`, during which the possible bug was fixed (indeed, the comment was removed after the refactoring). On the basis of the case discussed above, we further investigated to what extent the classes refactored during the implementation of new features contain a self-admitted technical debt. To this aim, we adopted the following procedure:

- Given a class C_i refactored in a release rel_j, we mined all the commits $c_1, c_2, ..., c_n$ between rel_{j-1} and rel_j and we extracted the source code of C_i in each commit c_i;
- For each version of C_i, we exploited `srcML` [30] to extract the comments from the Java code file.
- The set of retrieved comments was then analyzed to identify those reporting a self-admitted technical debt. To identify them, we exploited regular expressions to match inside comments the 62 self-admitted technical debt patterns defined by Potdar and Shihab [31].

From our analysis, we observed that 46% of the refactored classes contained a self-admitted technical debt in its previous versions. Interestingly, in 67% of the commits where a refactoring was performed the mention to the technical debt disappeared. Thus, we can affirm that in a good percentage of the cases developers apply refactoring during the implementation of new features in order to remove a technical debt previously left in the code. However, such refactoring operations do not tend to improve the overall readability of the source code. Indeed, we have observed that just 13% of classes refactored when implementing new features show an overall 30% improvement in terms of readability (as measured using the Buse and Weimer tool [20]).

A second relevant example is represented by the class `DiagramMemberFilePersister`, belonging to the package `org.argouml.persistence` of the ArgoUML project. Here, in version 0.32 the comment associated to the method `save` highlighted a *requirement debt* [29], [31]:

```
    // TODO: We need the project
  specific diagram settings here
```

In the subsequent version of the system (*i.e.,* version 0.34), during the implementation of a new methodology to save the UML diagrams, developers re-organized the source code by applying an *extract method* refactoring, paying off the debt by solving the settings issue previously raised during the implementation of the new feature. Also in this case, the cohesion of the class increases after the refactoring (*i.e.,* the LCOM decreases from 7 to 4). Even more evident is the case of the class `xerces.dom.AttributeMap` of the Apache Xerces system, where in version 1.4.1 the method `reconcileDefaults` was commented as follow:

```
    /** COMMENTED OUT!!!!!!!
  ******** Doing this dynamically
  is a killer, since editing the
  DTD isn't even supported this is
  commented out at least for now.
  In the long run it seems better
  to update the document on user's
  demand after the DTD has been
  changed rather than doing this
  anyway.*/
```

In the subsequent version (*i.e.,* version 1.4.2), the method was fixed when an update of the class needed to implement a new way to map the attributes of an XML file given as input to the system. Specifically, the developers applied a *replace method with method object* refactoring, giving to the method a specific responsibility and, thus, improving the adherence to the object-oriented programming principles.

In conclusion, we have strong indications that refactoring is related to feature introduction modification because developers adapt the source code before implementing new features, by applying refactoring types mainly concerned with the improvement of code cohesion and the adherence to the object-oriented programming principles. Moreover, several times refactoring is applied to remove previous technical debt introduced by developers to speed up the release process. Also in this case, our findings revealed that technical debt actually represents a strong motivation for refactoring the source code.

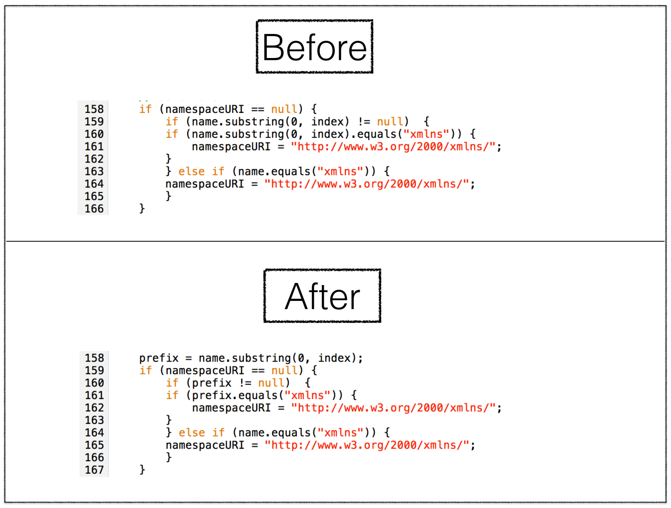
Observation 2. During the implementation of new features, developers tend to re-organize the source code through refactoring operations aimed at improving code cohesion and the adherence to the object-oriented programming principles. A deeper analysis revealed that one of the main reasons pushing developers to refactor source code is given by the presence of technical debt, which is payed off before introducing new code.

GM Model. The results for this model are shown in Table III. Although 75% of the statistically significant ORs are higher than one, it is worth noting that (i) often such values are just slightly higher than one (*e.g.*, the OR for the *move method* refactoring is 1.02 on Apache Ant), and (ii) the trends are not always consistent among the three projects considered. Thus, in general we can observe that refactoring operations involving

TABLE IV: Summary of the results achieved.

Change Type	Top Refactoring Operations	Underlying Reasons
Fault Repairing Modifications	add parameter consolidate duplicate conditional fragments move field remove assignment to parameters replace magic number with constant and replace nested cond guard clauses	Improving comprehensibility and maintainability of source code before fixing a bug.
Feature Introduction Modifications	add parameter extract method replace data with object replace method with method object	Removing technical debts previously left in the source code.
General Maintenance Operations	introduce explaining variable rename method	Improving source code readability and documentation.

Fig. 2: Method `synchronizeData` of the APACHE XERCES project before and after the refactoring operation applied.



```
158      if (namespaceURI == null) {
159          if (name.substring(0, index) != null)  {
160          if (name.substring(0, index).equals("xmlns")) {
161              namespaceURI = "http://www.w3.org/2000/xmlns/";
162          }
163          } else if (name.equals("xmlns")) {
164          namespaceURI = "http://www.w3.org/2000/xmlns/";
165          }
166      }
```

```
158      prefix = name.substring(0, index);
159      if (namespaceURI == null) {
160          if (prefix != null)  {
161          if (prefix.equals("xmlns")) {
162              namespaceURI = "http://www.w3.org/2000/xmlns/";
163          }
164          } else if (name.equals("xmlns")) {
165          namespaceURI = "http://www.w3.org/2000/xmlns/";
166          }
167      }
```

the modification of the system structure (*e.g., pull up/down field* refactoring) are not performed by developers when applying general maintenance modifications to the source code. On the other hand, there are two interesting cases regarding the *introduce explaining variable* and *rename method* refactoring operations that are worth discussing. In these cases, the ORs are high over all the systems indicating that classes involved in a large number of general modifications have a higher chance of being subject of refactoring operations aimed at improving their documentation as well as the quality of identifiers. The Mann-Whitney U test also revealed that the number of refactoring operations of these types applied during general modifications is statistically higher than other types of refactoring operations. For instance, between versions 1.4.1 and 1.4.2, the class `xerces.dom.DeferredAttrNSImpl` of the APACHE XERCES project was constantly refactored with the aim of improving its understandability. Indeed, developers applied a number of *introduce explaining variable* refactoring operations to make the roles of internal variables explicit. It is worth remarking that this refactoring is concerned with putting a result of an expression in a temporary variable with a name that explains the purpose [1].

An example is reported in Fig. 2, where a snippet of the code belonging to the method `synchronizeData` is depicted. In version 1.4.1 (upper side of Fig. 2), the `if` statements in lines #159 and #160 call the method `name.substring(0, index)` to extract the prefix of the qualified name of an XML file. In the subsequent version (lower side of figure 2), the developers introduced the variable `prefix` to capture the prefix of the qualified name before using it in the subsequent statements.

To further verify our conjecture about the goals of refactoring operations made by developers during general maintenance activities (*i.e.,* improvement of source code documentation), also in this case we conducted an additional analysis to understand whether such refactoring operations actually improved the readability of the source code. As previously done, we exploited the Buse and Weimer readability tool [20], observing that in 87% of the cases the classes refactored experienced an improvement in terms of readability, with an average improvement of 30%. These findings strengthen our hypotheses and allow use to conclude that the main reason pushing developers to refactor the source code while applying general modifications is improving the comprehensibility of classes.

Observation 3. When involved in general maintenance modifications, developers try to improve the comprehensibility of the source code by applying refactoring operations such as the *introduce explaining variable* and *rename method*. Furthermore, we observed the beneficial effects of refactoring on the overall readability of the refactored classes.

To summarize the results of our study, Table IV reports the achieved findings, indicating for each change type taken into account in our study (i) the top refactoring operations found through the quantitative analysis, and (ii) the main reasons why developers applied that refactoring operations, as pointed out by our qualitative investigation.

IV. THREATS TO VALIDITY

This section discusses the threats that could affect the validity of our study.

Construct Validity. One threat in this category regards the accuracy of the technique used to classify the types of changes analyzed. Indeed, we relied on the lexical approach proposed by Mockus *et al.* [22] which shows good performance.

However, we cannot exclude errors in the classification. A similar issue regards the tool employed to detect self-admitted technical debt [31]: as recently reported by Maldonado *et al.* [32], the lexical patters used by the tool suffer low recall rates: as a consequence, the results on the relationship between feature introduction changes and self-admitted technical debt are likely to be an under-estimation. Still in this category, it is worth mentioning possible issues due to the quality of the dataset exploited. As reported by Bavota *et al.* [18], the refactoring operations have been manually validated after a first detection performed using a refactoring detector named REFFINDER [33]. Despite this, it is known that REFFINDER (i) cannot deal with multiple refactoring operations performed within one commit [34], and (ii) is not able to identify some refactoring types (*e.g.,* Extract Class refactoring) [33]. Thus, our study is limited to the refactoring operations actually detectable by using REFFINDER and for which a manual validation aimed at reducing possible imprecisions was previously conducted by Bavota *et al.* [18]. As a consequence, we believe that our study is conducted upon a dataset having a good degree of data quality. Finally, the refactoring operations in the exploited dataset have been detected at release-level, while the different change types have been identified at commit-level. While the different granularity could have influenced our observations, it is important to note that all the systems analyzed tend to frequently issue releases (as the reader can see from the number of considered releases): thus, the analysis at release level is not necessarily *coarse-grained*. Moreover, we mitigated this threat by conducting qualitative analyses aimed at illustrating the reasons why specific refactoring operations helped during the development of a given change type.

Conclusion Validity. To assess the relationships between different types of changes and refactoring operations, we exploited logistic regression models, being sure to avoid unreliable results by just considering the refactoring operations applied more than ten times over the change history of the systems considered. Moreover, other than highlighting cases of significant correlations, we reported and discussed OR values.

Internal Validity. There are factors that might have influenced our observations. Indeed, we evaluated different types of changes without considering the development type adopted by the projects in our study, as well as their life cycle or their development activity. However, this is an exploratory study on the relationships between changes and refactoring. Future effort will be devoted to the analysis of the co-factors mentioned above.

External Validity. While the study is limited when considering the number of projects (3), it is worth noting that we evaluated 12,922 refactoring operations spread across 63 releases (for a total of 30 years of development). Moreover, we considered open source systems for our analysis, since the source code of commercial ones is not available. Future investigations aimed at corroborating our findings are desirable.

V. RELATED WORK

In the recent past, the research community spent a lot of effort in devising tools for suggesting refactoring operations as well as to understand under which circumstances developers refactor and which are the relationships between quality and refactoring (e.g., [35]). Due to the empirical nature of this paper, in the following we summarize the previous work aimed at empirically characterizing the refactoring activities performed by developers. A complete report of the automatic techniques able to suggest refactoring operations is available in [36].

Wang *et al.* [17] reported the results of a survey carried out with 10 industrial programmers where the goal was to identify the major factors pushing developers in performing refactoring. They identified 12 main factors, classifying them in *intrinsic* and *external* motivators. Specifically, the former category is composed of the factors related to external rewards (*e.g.,*, the *Responsibility with Code Authorship* represents an intrinsic motivator, since developers want to program high quality code). As for external motivators, a clear example is represented by the *Recognitions from Others*, *i.e.,* gain recognitions from others. Our study is complementary to the one by Wang *et al.*, since it shows which are the typical refactoring operations applied during different types of changes.

Murphy-Hill *et al.* [14] studied how developers perform refactoring, analyzing eight large scale software systems. Key findings of this study are that (i) 41% of programming activities contain refactoring traces, (ii) developers generally do not configure refactoring tools, (iii) commit messages cannot predict refactoring activities over the history of a software system because developers tend to not explicit refactoring activities when writing commit messages, (iv) most of the refactoring is *floss*, *i.e.,*, applied within other development activities, and (v) almost all the refactoring operations are done manually by developers without the help of any tool.

Complementary to the work by Murphy-Hill *et al.* is the paper by Kim *et al.* [37], who performed a survey with 328 software engineers of Microsoft in order to investigate (i) when and how they do refactoring, (ii) whether automated tools are used to support refactoring operations, and (iii) the developers' opinion on the benefits, risks, and challenges of refactoring [37]. In the first place, the important result achieved was about the perception of refactoring as a non behavior-preserving activity: indeed, almost 50% of developers fear that refactoring can introduce side-effects such as the bugs. Moreover, the main motivation that push developers in applying refactoring is the poor readability of the source code, while at least 51% of the participants declared that he/she usually perform refactoring manually.

Kim *et al.* [37] also conducted an analysis on the change history of the Windows 7 system, reporting that classes subject of refactoring activities experienced a notable reduction in terms of number of inter-module dependencies and post-release defects with respect to other modules. Similar results have been obtained by Kataoka *et al.* [38] and Gatrell and

Counsell [39]. In the first study, the authors analyzed the history of an industrial software system comparing the classes subject to the application of refactoring operations with the classes never refactored. They observed a decrease of coupling metrics. Regarding the work by Gatrell and Counsell, they conducted an empirical study aimed at quantifying the effect of refactoring on change- and fault-proneness of classes. The authors monitored a commercial C# system for twelve months identifying the refactoring operations applied during the first four months. They examined the same classes for the second four months in order to determine whether the refactoring results in a decrease of change- and fault-proneness. They also compared such classes with the classes of the system that, during the same time period, have not been refactored. Results revealed that classes subject to refactoring have a lower change- and fault-proneness, both considering the time period in which the same classes were not refactored and classes in which no refactoring operations were applied.

Finally, it is worth discussing the studies that focused on the relationship between refactoring and software quality. In particular, Bavota *et al.* [26] investigated the extent to which refactoring activities may induce faults. They show that specific types of refactorings that involve hierarchies (*e.g., pull down method*) can often induce faults. On the other hand, refactoring having as goal the re-location of source code (*e.g., move method*) are likely to be harmless. Bavota *et al.* [18] also conducted a study aimed at understanding the relationships between code quality and refactoring. In particular, they studied the evolution of 63 releases of 3 open source systems in order to investigate the characteristics of code components increasing/decreasing their chances of being object of refactoring operations. Results indicate that often refactoring is not performed on classes having a low metric profile, while almost 40% of the times refactoring operations have been performed on classes affected by smells. However, just 7% of them actually removed the smell. While we share with this work the dataset of refactoring operations used to run our study, we also demonstrated that different types of changes better explain refactoring operations performed by developers.

Silva *et al.* [16] monitored a large set of Java projects in order to identify the refactoring operations applied by developers, and then they asked the developers to explain the reasons behind their decision to refactor the code. They found that refactoring is mainly driven by changes in the requirements rather than the presence of code smells. On the one hand, our findings qualitatively confirm the results by Silva *et al.*, however they also show that in a good percentage of the cases technical debts and duplicate code can be the causes of the activities performed by developers to re-organize the source code.

Stroggylos and Spinellis [40] studied the impact of refactoring operations on the values of eight object-oriented quality metrics. Their results show the possible negative effects that refactoring can have on some quality metrics (*e.g.,* increased value of the LCOM metric). On the same line, Stroullia and Kapoor [41], analyzed the evolution of one system observing a decrease of LOC and NOM (Number of Method) metrics on the classes in which a refactoring has been applied. Szoke *et al.* [6] performed a study on five software systems to investigate the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality. Alshayeb [4] investigated the impact of refactoring operations under five different perspectives, *i.e.,* adaptability, maintainability, understandability, reusability, and testability. Their main findings showed that refactoring provides benefits on some classes, but at the same time such benefits are counterbalanced by a decrease of code quality in other classes. Moser *et al.* [5] investigate the impact of refactoring on agile developers' productivity in industry. They found that on the one hand refactoring increases software quality, and on the other hand provides benefits in term of productivity. In the context of this study, we somehow confirmed the ability of refactoring in improving non-functional attributes of the source code (*e.g.,* by increasing the readability of refactored classes).

VI. Conclusion

Refactoring is widely recognized as an activity able to improve software quality [17] and providing other beneficial effects, such as developers' productivity [5]. Previous empirical studies that have assessed the motivations behind the application of refactoring based on developers' opinion [15], [16], found that refactoring is mainly driven by changes in the requirements rather than by software quality. While such studies are based on the developers' opinions, no investigations based on the analysis of software repositories have confirmed such findings. To this aim, we verified whether and to what extent refactoring is driven by different types of changes, *i.e.*, *Fault Repairing Modification*, *General Maintenance Modification*, and *Feature Introduction Modification*, applied over the change history of three software systems.

The results of the study indicate that classes experiencing a higher number of bug fixing activities are more subject to operations that improve their maintainability and comprehensibility, while classes where the number of new features implemented is higher are more prone to be refactored with regard to code cohesion and adherence to the object-oriented programming principles. The underlying reasons behind the application of such refactoring operations fall into the presence of duplicate code or of previously self-admitted technical debts. Thus, in most cases changes are associated with the payment of an existing accumulated debt. Finally, general maintenance modifications lead to refactoring aimed at improving comprehensibility and identifier quality, leading to an overall improvement of the readability of the source code.

Other than corroborating our results on a larger number of systems, our future research agenda includes (i) a deeper investigation into the benefits provided by refactoring operations applied by developers in different situations, and (ii) the definition of predictive models able to suggest developers about which type of refactoring should be applied in a given situation.

REFERENCES

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

[2] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits," in *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 504–507.

[3] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[4] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319 – 1326, 2009.

[5] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*, B. Meyer, J. R. Nawrocki, and B. Walter, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 252–266.

[6] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2014, pp. 95–104.

[7] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[8] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Trans. Softw. Eng.*, 2015.

[9] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Proc. Int'l Conf on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, 2009.

[11] 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.

[12] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, 2014.

[13] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, p. 4, 2014.

[14] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, 2011.

[15] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. ACM, 2012, p. 50.

[16] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proc. Int'l Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 858–870.

[17] Y. Wang, "What motivates software engineers to refactor source code? evidences from professional developers," in *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 2009, pp. 413–416.

[18] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[19] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 78–88.

[20] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 546–558, 2010.

[21] I. Kadar, P. Hegedus, R. Ferenc, and T. Gyimthy, "A code refactoring dataset and its assessment regarding software maintainability," in *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 599–603.

[22] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2000, pp. 120–.

[23] D. Hosmer and S. Lemeshow, *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

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

[25] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[26] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2012, pp. 104–113.

[27] B. Du Bois, S. Demeyer, and J. Verelst, "Does the "refactor to understand" reverse engineering pattern improve program comprehension?" in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2005, pp. 334–343.

[28] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[29] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[30] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *Proc. Int'l Workshop on Program Comprehension (IWPC)*. IEEE, 2003, pp. 134–143.

[31] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proc. Int'l Conf on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 91–100.

[32] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[33] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[34] Q. D. Soetens, J. Pérez, S. Demeyer, and A. Zaidman, "Circumventing refactoring masking using fine-grained change recording," in *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2015, pp. 9–18.

[35] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2012, pp. 29–38.

[36] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. Springer, 2014, pp. 387–419.

[37] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, 2014.

[38] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2002, pp. 576 – 585.

[39] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Sci. Comput. Program.*, vol. 102, no. C, pp. 44–56, 2015.

[40] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07. IEEE, 2007, pp. 10–.

[41] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *OOIS 2001*, X. Wang, R. Johnston, and S. Patel, Eds. Springer, 2001, pp. 113–122.