# An Experimental Investigation on the Innate Relationship between Quality and Refactoring

Gabriele Bavota[1], Andrea De Lucia[2], Massimiliano Di Penta[3], Rocco Oliveto[4], Fabio Palomba[2]

[1] *Free University of Bozen-Bolzano, Bolzano, Italy,* [2] *University of Salerno, Fisciano (SA), Italy*
[3] *University of Sannio, Benevento, Italy,* [4] *University of Molise, Pesche (IS), Italy*

**Abstract**

Previous studies have investigated the reasons behind refactoring operations performed by developers, and proposed methods and tools to recommend refactorings based on quality metric profiles, or on the presence of poor design and implementation choices, i.e., code smells. Nevertheless, the existing literature lacks observations about the relations between metrics/code smells and refactoring activities performed by developers. In other words, the characteristics of code components increasing/decreasing their chances of being object of refactoring operations are still unknown. This paper aims at bridging this gap. Specifically, we mined the evolution history of three Java open source projects to investigate whether refactoring activities occur on code components for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactoring operations. Results indicate that, more often than not, quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.

*Keywords:* Refactoring, Code Smells, Empirical Study

## 1. Introduction

Refactoring has been defined by Fowler as "*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*" [21]. This definition entails a strong relationship between refactoring and internal software quality, i.e., refactoring improves software quality (*improves the software internal structure*). This has motivated research on bad smell and antipattern detection and on the identification of refactoring oppotunities [54, 34, 43, 20, 5, 9, 12, 26].

However, whether refactoring is actually guided by poor design has not been empirically evaluated enough. Thus, this assumption still remains—for some aspects—a common wisdom that has generated controversial positions [27]. Specifically, there are no studies that **quantitatively** analyze which are the quality characteristics of the source code increasing their likelihood of being subject of refactoring operations. To the best of our knowledge, the available empirical evidence is based on two surveys performed with developers trying to understand the reasons why developers perform refactoring operations [55, 27].

In addition, concerning the improvement of the internal quality of software, empirical studies have only shown that generally refactoring operations improve the values of quality metrics [25, 28, 46, 37, 49], while the effectiveness of refactoring in removing design flaws (such as code smells) is still unknown.

In order to fill this gap, we use an existing tool, namely Ref-Finder [45], to automatically detect refactoring operations of 52 different types on 63 releases of three Java software systems, namely Apache Ant[1], ArgoUML[2], and Xerces-J[3]. Since Ref-Finder can identify some false positives, we manually analyzed the 15,008 refactoring operations detected by the tool. Among them, 2,086 were classified as false positives. Thus, in the context of our study we analyzed 12,922 refactoring operations.

Having identified the refactoring operations, for each class in the analyzed systems' releases we (i) measured a set of eleven quality metrics, and (ii) detected if it is affected by any instance of eleven code smells. Using these data we verify whether refactoring operations occur on code components for which the factors above (i.e., quality metrics, presence of code smells) suggest there might be need for refactoring operations. In addition, we also measure the effectiveness of refactoring operations in terms of their ability to remove code smells.

The results achieved can be summarized as follows:

1. More often than not, quality metrics do not show a clear relationship with refactoring. In other words

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

Table 1: Characteristics of the analyzed projects

| Project | Period | Analyzed | #Releases | Classes | KLOC |
|---|---|---|---|---|---|
| Apache 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** | - | - |

quality metrics might suggest classes as good candidates to be refactored that are generally not involved in developers' refactoring operations.

2. Among the 12,922 refactoring operations analyzed, 5,425 are performed by developers on code smells (42%). However, of these 5,425 only 933 actually remove the code smell from the affected class (7% of total operations) and 895 are attributable to only four code smells (i.e., Blob, Long Method, Spaghetti Code, and Feature Envy). Thus, not all code smells are likely to trigger refactoring activities.

In summary, such results suggest that (i) more often than not refactoring actions are not a direct consequence of worrisome metric profiles or of the presence of code smells, but rather driven by a general need for improving maintainability, and (ii) refactorings are mainly attributable to a subset of known smells. For all these reasons, the refactoring recommendation tools should not only base their suggestions on code characteristics, but they should consider the developer's point-of-view in order to propose meaningful suggestions of classes to be refactored.

The paper is organized as follows. Section 2 describes the design of our empirical study, while Section 3 reports and discusses the obtained results. Section 4 analyzes and discusses the threats that could affect the results of our study. After a discussion of the related literature (Section 5), Section 6 concludes the paper.

## 2. 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 (i) if quality metrics and code smells presence provide indications on which code components are more/less likely of being refactored; and (ii) as a consequence, to what extent are refactoring operations effective in removing code smells from source code. The object systems, the tools, and the raw data are available for replication in our online appendix.[4]

### 2.1. Context and Research Questions

The study aims at addressing the following research questions:

- **RQ**$_1$: Are refactoring operations performed on classes having a low-level of maintainability as indicated by quality metrics?
- **RQ**$_2$: To what extent are refactoring operations (i) executed on classes exhibiting code smells and (ii) able to remove code smells?

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, while Xerces-J is a XML parser for Java. Although this looks a relatively small context (three projects only), such a choice has been necessary to allow us manually validating the detected refactoring and code smells, as detailed below. Table 1 reports characteristics of the analyzed systems, namely analyzed releases, number of analyzed releases, and size range (in terms of KLOC and # of classes).

### 2.2. Study Variables and Data Extraction

The **dependent variables** considered in our study, for all the research questions, are the refactoring operations (of different types) being observed across releases of different programs. The **independent variables** are the factors we relate to such observed refactoring and namely:

1. For **RQ**$_1$, a series of quality metrics (described below).
2. For **RQ**$_2$, the presence of code smells (of different types) in software releases.

To answer our research questions, we first need to detect refactorings over the evolution history of the studied systems. To this aim we use an existing tool, Ref-Finder [45], to detect refactoring operations performed between each subsequent couples of releases of each system. Ref-Finder has been implemented as an Eclipse plug-in and it is able to detect 63 different kinds of refactoring operations. 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% [45]. 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 refactoring operations identified by Ref-Finder. Specifically, each refactoring operation identified by the tool was manually analyzed through source code inspection by two Master's students from the University of Salerno. The students individually validated each of the proposed refactoring operations.

Once students validated the refactoring operations, 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*. Of the 15,008 refactoring operations detected by Ref-Finder, 12,922 operations have been manually classified as actual refactoring operations, producing as output 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. Table 2 reports the number of refactoring operations (as well as the

[4] `http://dx.doi.org/10.6084/m9.figshare.1207916`

Table 2: 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** |

number of different types of refactorings) identified on the three systems after the manual validation. While the extracted refactoring operations are needed to answer all our research questions, in the following we detail on data collection activities made to specifically answer each research question.

#### 2.2.1. Data Extracted to Answer $\boldsymbol{RQ}_1$

To answer $\mathbf{RQ}_1$, we need to measure—for each class of the analyzed systems—a set of quality metrics. Specifically, we measure for each class in the analyzed systems' releases a set of eleven quality metrics. Since we know in each release which classes have been subject of which refactoring operations, we can use these metrics to understand if any of them suggest that the considered classes need to be refactored.

The employed quality metrics are reported in Table 3. Our choice of the metrics is not random. We considered LOC since it has been demonstrated to be one of the better metrics in predicting the number of faults in a code component [18]. Thus, it is also possible that LOC also helps in identifying classes having a poor design from the developers point of view. The Chidamber & Kemerer (CK) metrics [17] have been object of several empirical studies showing their ability of capturing different aspects of code maintainability [3, 10, 13, 14, 17, 23, 29]. We also adopted NOA and NOO since they measure quality aspects of a class that are not taken into account by the CK metrics (see Table 3). Finally, we also considered semantic metrics since (i) they have been shown to not correlate with structural metrics [31] and (ii) in a recent study [7] the Conceptual Coupling Between Classes (CCBC) has been shown to be the coupling metric better capturing the developers perception of coupling between code components. To extract these metrics, we developed a tool exploiting the Eclipse JDT API to extract all needed information from source code.

#### 2.2.2. Data Extracted to Answer $\boldsymbol{RQ}_2$

To answer $\mathbf{RQ}_2$, we analyze each class of the 63 considered software releases to verify if it is affected by any code smell. In particular, we detected instances of the eleven code smells reported in Table 4 defined by Fowler [22] and Brown *et al.* [15]. Also in this case, the goal is to understand if the presence of specific code smells increases/decreases the changes of the affected code components of being the object of refactoring actions. To detect the code smells we developed a simple tool that outputs a list of candidate classes potentially exhibiting a code smell. Then, we manually validated the candidate code smells suggested by the tool. The validation was performed by a Master and a Ph.D. student, who individually analyzed and classified as *true positive* or *false positive* all candidate code smells. Finally, the students performed an open discussion with researchers to resolve any conflicts and reach a consensus on the detected code smells.

Table 6: Number of smells analyzed in this paper

| Project | #Smells | Distinct types of smells |
|---|---|---|
| Apache Ant | 1,493 | 10 |
| ArgoUML | 1,197 | 7 |
| Xerces-J | 2,788 | 10 |
| **Overall** | **5,478** | **10** |

To ensure high recall, our tool uses very simple detection rules that overestimate the presence of code smells in the code. This is done at the expense of precision. Even though this choice resulted in a longer list of candidates and thus in a more expensive manual validation, it was necessary to study the real distribution of code smells in the analyzed releases. Table 5 reports the rules applied by our tool to detect each of the eleven analyzed code smells. Note that we choose not to use existing detection tools [34, 20, 54] because (i) none of them has ever been applied to detect all the studied code smells, and (ii) their detection rules are generally restrictive to ensure a good compromise between recall and precision, thus they may miss some code smell instances. Table 6 reports the number of code smells and the number of different types of smells identified on the three systems after the manual validation.

Knowing the list of classes affected by each code smell in each software release, we are also able to verify to what extent refactoring operations are able to remove code smells from source code. In particular, given a refactoring operation (e.g., Extract Class) $o_i$ performed in a release $r_j$ on a class affected by a code smell (e.g., Blob class) $a_k$, we can verify if $o_i$ was able to remove $a_k$ by checking if $a_k$ is still present in the release $r_{j+1}$ (and thus, the code smell has not be removed) or not (the code smell has been removed).

### 2.3. Analysis Method

To address the two research questions formulated above, we build, for each object system and for each kind of refactoring operation performed on it, logistic regression models relating a (dichotomous) dependent variable—indicating whether or not a particular type of refactoring was performed—with independent variables represented by the quality indicators (metrics, and presence of code smells) described above. Logistic regression models [24] relate dichotomous dependent variables with one or more independent variables as follows:

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

where $X_i$ are the independent variables describing the phenomenon, and $C_i$ the coefficients (estimates) of the logis-

Table 3: Quality metrics measured to answer $\mathbf{RQ}_1$

| Metric | Description |
|---|---|
| Lines of Code (LOC) | The number of lines of code excluding white spaces and comments |
| Weighted Methods per Class (WMC) [17] | The complexity of a class as the sum of the McCabe's cyclomatic complexity of its methods |
| Depth of Inheritance Tree (DIT) [17] | The depth of a class as the number of its ancestor classes |
| Number Of Children (NOC) [17] | The number of direct descendants (subclasses) of a class |
| Response for a Class (RFC) [17] | The number of distinct methods and constructors invoked by a class |
| Coupling Between Object (CBO) [17] | The number of classes to which a class is coupled |
| Lack of COhesion of Methods (LCOM) [17] | The higher the pairs of methods in a class sharing at least a field, the higher its cohesion |
| Number of Operations Added by a subclass (NOA) [30] | The number of methods added by a subclass to the methods inherited by its superclass |
| Number of Operations Overridden by a subclass (NOO) [30] | The number of methods overridden by a subclass among those inherited by its superclass |
| Conceptual Coupling Between Classes (CCBC) [44] | The higher the textual similarity between two classes, the higher their coupling |
| Conceptual Cohesion of Classes (C3) [31] | The higher the textual similarity between the methods of a class, the higher its cohesion |

Table 4: Code smells detected to answer $\mathbf{RQ}_3$

| Name | Description |
|---|---|
| Class Data Should Be Private (CDSBP) [15] | A class exposing its fields, violating the principle of data hiding. |
| Complex Class [15] | A class having at least one method having a high cyclomatic complexity. |
| Feature Envy [22] | A method is more interested in a class other than the one it actually is in. |
| Blob Class (Blob) [15] | A large class implementing different responsibilities and centralizing most of the system processing. |
| Lazy Class [22] | A class having very small dimension, few methods and with low complexity. |
| Long Method [22] | A method that is unduly long in terms of lines of code. |
| Long Parameter List (LPL) [22] | A method having a long list of parameters, some of which avoidable. |
| Message Chain [22] | A long chain of method invocations is performed to implement a class functionality. |
| Refused Bequest [22] | A class redefining most of the inherited methods, thus signaling a wrong hierarchy. |
| Spaghetti Code [15] | A class implementing complex methods interacting between them, with no parameters, using global variables. |
| Speculative Generality [22] | A class declared as abstract having very few children classes using its methods. |

tic regression model. We used the *R* statistical software (`http://www.r-project.org/`) to build the logistic regression models. Specifically, we built the following two models:

1. *Metrics.* The first model uses the eleven measured quality metrics as independent variables and the application of the particular type of refactoring (e.g., Extract class) as the dependent variable. All metrics have been normalized using the z-score, i.e., by subtracting the mean and dividing by the standard deviation.
2. *Smells.* The second model uses the presence of the considered code smells in a class as independent (and Boolean) variables, and the application of the particular type of refactoring (e.g., Extract class) as the dependent variable.

Note that, given a refactoring type $r_i$ and a system $s_j$, we build the two models presented above only if the refactoring type $r_i$ has been applied on the system $s_j$ at least 10 times. This is done to avoid the creation of unreliable logistic regression models.

We are aware that our models could be affected by multi-collinearity [39], which occurs when two or more independent variables are highly correlated and can be predicted one from the other, possibly affecting the resulting model. We assess our models for the presence of multi-collinearity in two different ways:

1. Whenever possible, i.e., for the models based on metrics, we compute the Spearman's rank correlation between all possible pairs of metrics, to determine whether there are pairs of strongly correlated metrics (i.e., with a Spearman's $\alpha > 0.8$). If two independent variables are highly correlated, one of them should be removed from the model.
2. By using a stepwise variable removal procedure based on the Companion Applied Regression (*car*) R package[5], and in particular based on the *vif* (variance inflation factors) function [39].

Once we have avoided multi-collinearity using the procedure described above, we build the logistic regression models with the variables remained after the pruning. Then, for each model we analyze (i) whether each independent variable is significantly correlated with the dependent variable (we consider a significance level of $\alpha = 5\%$), and (ii) we quantify such a correlation using the Odds Ratio (OR) [50] which, for a logistic regression model, is given by $e^{C_i}$. The higher the OR for an independent variable, the higher its ability to explain the dependent variable. However, the interpretation of the OR changes between the two kinds of models we built, due to the different measurement scale of the independent variables, i.e., ratio for the metric-based model and nominal (categorical) for the code smell-based model. In particular, for the model built using quality metrics, the OR for an independent variable indicates the increment of chances for a class to be subject of refactoring in consequent of a one-unit increase of the independent variable. For example, if we found that the CBO has an OR of 1.15 when building a logistic regression model for the Extract Class refactoring operation, this means that for each one-unit increase of the CBO value for a class, it has 15% higher chances of being involved in an Extract

[5] http://cran.r-project.org/web/packages/car/index.html

Table 5: The rules used by our tool to detect candidate code smells

| Name | Description |
|---|---|
| Class Data Should Be Private | A class having at least one public field. |
| Complex Class | A class having at least one method for which McCabe cyclomatic complexity is higher than 10. |
| Feature Envy | All methods having more calls with another class than the one they are implemented. |
| Blob Class | All classes having (i) cohesion lower than the average of the system AND (ii) LOCs > 500. |
| Lazy Class | All classes having LOCs lower than the first quartile of the distribution of LOCs for all system's classes. |
| Long Method | All methods having LOCs higher than the average of the system. |
| Long Parameter List | All methods having a number of parameters higher than the average of the system. |
| Message Chain | All chains of methods' calls longer than three. |
| Refused Bequest | All classes overriding more than half of the methods inherited by a superclass. |
| Spaghetti Code | A class implementing at least two long methods (see previous rule) interacting between them through method calls or shared fields. |
| Speculative Generality | A class declared as abstract having less than three children classes using its methods. |

Class refactoring operation. On the other side, for the model built using code smells, the OR indicates the likelihood of a class affected by a code smell of being involved in refactoring operations with respect to a non-affected class. As example, if we found that the code smell Blob has an OR of 3 when building a logistic regression model for the Extract Class refactoring operation, this means that classes affected by the Blob code smell have 3 times higher chances of being involved in an Extract Class refactoring operation than classes not affected by it.

Finally, to verify the ability of refactoring in removing code smells from source code, we simply analyze for each refactoring type (e.g., Extract class) the percentage of times it is able to remove each type of code smell (e.g., Blob class).

## 3. Empirical Study Results

This section discusses the results of our study, aimed at addressing the research questions formulated in Section 2.1. As explained in Section 2.3, before building the logistic regression models, we performed a multi-collinearity analysis. As a result of such analysis, we found that:

- For the models based on metrics, and only for the Xerces project, the stepwise regression procedure removed the DIT metric from the logistic regression model. Consistently with that, we found a strong ($\alpha = 0.83$) Spearman's rank correlation between DIT and NOA. This is not entirely surprising as both DIT and NOA capture information related to inheritance relations between classes. No multi-collinearity was found for the other two projects (Apache Ant and ArgoUML).
- For the models based on smells, no independent variable is affected by multi-collinearity.

### 3.1. *Are refactoring operations performed on classes having a low-level of maintainability as indicated by quality metrics?*

Table 7 reports the ORs of quality metrics obtained when building a logistic regression model for data concerning each refactoring operation. Statistically significant values, i.e., those for which the $p$-value is lower than 0.05, are reported in **bold** face. In the following, we will mainly focus our discussion on such statistically significant values.

First, we can immediately notice that longer classes (in terms of LOC) generally have a higher chance of being involved in a refactoring operations (the ORs for LOC are higher than 1 in 71% of significant ORs). This is quite an expected result. More interesting are the results—and in particular the observed OR values—for the other metrics.

The WMC metric of a class, i.e., the sum of the McCabes' cyclomatic complexity of its methods, exhibits very high ORs for some of the refactoring operations dealing with the simplification of methods inside a class. However, this is not always true for all systems. In particular, classes having high WMC have:

- In ApacheAnt (OR 22.35), a much higher chance of being involved in a *consolidate conditional expression* refactoring, performed to simplifying a sequence of conditional expressions which produce the same result by combining them into a single expression. The OR for WMC on this refactoring is also very high on ArgoUML (9.54), even if not statistically significant.
- In ApacheAnt (OR 5.47), a higher chance of being involved in a *remove control flag* refactoring, performed to replace a variable that is acting as a control flag for a series of Boolean expressions with a simpler break statement. In this case, also on the other systems the OR is higher than 1, but not statistically significant.
- In ApacheAnt (OR 8.9), a higher chance of being involved in a *replace nested conditional with guard clauses* refactoring, applied to methods in which the conditional behavior does not make clear the normal path of execution. Also in this case, on both other systems the OR is higher than 1, but not statistically significant.
- In Xerces (OR 9.94), a higher chance of being involved in an *inline temp* refactoring, performed to remove temporary variables that are only assigned once with a simple expression. Also in ApacheAnt the OR for this refactoring is high (3.55) but, again, not statistically significant.

Surprisingly, we did not find any statistically significant OR higher than one for WMC on models built for the

Table 7: Quality metrics model: OR of metrics when building logistic model. Statistically significant ORs are reported in bold face.

| Refactoring | System | LOC | WMC | DIT | NOC | RFC | CBO | LCOM | NOA | NOO | CCBC | C3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| add parameter | ApacheAnt | **3.51** | 0.40 | **0.39** | **1.24** | 1.61 | **0.41** | 1.26 | 0.59 | 0.98 | 0.75 | **0.22** |
| add parameter | ArgoUML | **1.15** | **1.01** | 1.62 | 1.09 | 1.45 | **0.79** | **0.04** | **1.04** | 0.69 | 0.34 | 0.54 |
| add parameter | Xerces | 1.06 | **2.26** | - | **1.13** | **0.67** | **1.10** | **0.66** | 1.31 | 0.99 | 0.30 | **0.56** |
| consolidate cond expression | ApacheAnt | 0.27 | **22.35** | **1.00** | **0.90** | **1.58** | **0.36** | **0.23** | 0.14 | 1.04 | 1.54 | **0.34** |
| consolidate cond expression | ArgoUML | 1.79 | 1.07 | 1.21 | **0.77** | **3.05** | **0.74** | **0.01** | **1.44** | 1.28 | 1.56 | 0.30 |
| consolidate cond expression | Xerces | 1.30 | 9.54 | - | 1.07 | 0.63 | 1.01 | 0.78 | 1.10 | 1.13 | 1.49 | 0.36 |
| consolidate duplicate cond fragments | ApacheAnt | 0.53 | 8.02 | 0.53 | **1.25** | **2.35** | **0.38** | 0.47 | **0.54** | **0.63** | 0.91 | **0.31** |
| consolidate duplicate cond fragments | ArgoUML | **1.09** | **1.92** | **1.23** | **1.02** | **2.62** | **1.35** | 0.00 | 0.53 | 1.29 | 0.54 | **0.72** |
| consolidate duplicate cond fragments | Xerces | 1.26 | 6.77 | - | **1.13** | **0.77** | **1.86** | 0.92 | 1.13 | 1.02 | 0.68 | 0.56 |
| extract method | ApacheAnt | **0.83** | 5.84 | 0.40 | 1.24 | **2.42** | 0.58 | **0.31** | **0.34** | 0.84 | 0.78 | 0.20 |
| extract method | ArgoUML | **1.29** | **0.27** | 1.55 | 1.03 | 2.40 | **1.12** | **0.05** | 0.94 | 0.86 | 0.35 | 0.28 |
| extract method | Xerces | **1.16** | 0.64 | - | **1.11** | 1.33 | **1.49** | 0.66 | 0.99 | 1.08 | 0.67 | **0.28** |
| extract superclass | ArgoUML | **2.56** | **0.36** | **0.45** | **1.11** | **0.65** | 1.04 | 0.00 | **0.68** | **0.38** | 0.95 | **0.06** |
| form template method | ArgoUML | **4.10** | **0.00** | 0.00 | 1.82 | **0.88** | 1.18 | **0.00** | **2.94** | **1.82** | 0.00 | **0.38** |
| inline method | ApacheAnt | 1.16 | **0.14** | 3.23 | 1.41 | **4.04** | **0.20** | 0.11 | 0.13 | 0.42 | 1.40 | 0.08 |
| inline method | ArgoUML | 1.15 | 0.63 | 3.46 | 1.15 | 1.46 | 1.35 | 0.24 | 0.84 | 0.85 | 0.17 | 0.30 |
| inline method | Xerces | 0.71 | 1.59 | - | **1.10** | **1.10** | **1.15** | **0.39** | 0.98 | **1.30** | **1.14** | **0.07** |
| inline temp | ApacheAnt | **1.56** | 3.55 | 0.57 | 1.14 | **0.59** | 0.98 | **0.85** | **0.35** | 0.63 | **0.95** | **0.29** |
| inline temp | ArgoUML | 1.17 | 0.96 | 1.58 | 0.43 | **1.02** | **1.31** | **0.16** | **0.78** | 0.94 | **0.82** | 0.29 |
| inline temp | Xerces | 1.80 | **9.94** | - | 1.09 | **0.61** | 1.64 | **0.68** | 1.37 | 1.00 | **0.97** | **0.70** |
| introduce assertion | ArgoUML | 0.13 | 0.68 | 6.97 | 0.68 | 6.23 | 1.83 | 0.02 | 0.27 | 0.00 | 0.83 | **0.39** |
| introduce explaining variable | ApacheAnt | 1.54 | **0.81** | 1.14 | **1.29** | 1.88 | 0.61 | 1.04 | 0.10 | 0.18 | 1.04 | 0.16 |
| introduce explaining variable | ArgoUML | 0.82 | 1.05 | **0.83** | 1.00 | 2.54 | 1.16 | 0.27 | 0.80 | 0.99 | **0.69** | 0.53 |
| introduce explaining variable | Xerces | 1.00 | 4.12 | - | 1.11 | 0.86 | **1.81** | **0.97** | 1.04 | 1.02 | 0.80 | **0.46** |
| introduce null object | ArgoUML | 0.42 | 1.90 | **1.12** | 0.97 | **0.00** | **2.21** | 0.00 | 4.91 | 2.53 | 1.52 | 0.92 |
| introduce parameter object | Xerces | 0.88 | 2.97 | - | 1.08 | **1.33** | 0.25 | 0.15 | 1.36 | 0.91 | 0.00 | **0.06** |
| move field | ApacheAnt | 7.53 | 0.02 | 2.80 | 1.35 | **6.82** | 0.12 | 1.51 | 0.43 | 0.47 | 0.65 | 0.23 |
| move field | ArgoUML | **10.40** | **0.00** | **1.58** | 0.92 | 0.77 | 1.16 | **0.00** | **0.75** | **0.24** | 1.08 | **1.72** |
| move field | Xerces | 1.07 | 2.63 | - | 1.00 | **0.61** | **1.58** | **0.89** | **1.04** | 1.10 | **0.55** | 0.10 |
| move method | ApacheAnt | 1.41 | 0.10 | 0.51 | 1.39 | **7.13** | 0.25 | **0.77** | 1.04 | 0.37 | **1.13** | **0.83** |
| move method | ArgoUML | **1.18** | **1.61** | **2.97** | 1.06 | **0.60** | 1.26 | **0.04** | **0.81** | **0.92** | **0.58** | **1.22** |
| move method | Xerces | 1.03 | 2.91 | - | **0.82** | **0.50** | 1.29 | **0.71** | 1.18 | 1.12 | **0.50** | **0.13** |
| pull up field | Xerces | **2.44** | 0.37 | - | **1.17** | 0.89 | **1.90** | **0.60** | 20.31 | 6.07 | **0.25** | **0.73** |
| pull up method | Xerces | 1.71 | 0.00 | - | 0.03 | 0.00 | 4.20 | 0.00 | 8.26 | 20.91 | **0.45** | 0.00 |
| push down field | Xerces | **5.49** | **0.22** | - | 2.34 | **0.06** | **0.86** | 0.44 | 0.31 | **1.64** | **0.23** | **1.43** |
| push down method | Xerces | 29.65 | 0.00 | - | 1.54 | 0.00 | **3.00** | **0.07** | **0.32** | 1.42 | **0.38** | **1.18** |
| remove assignment to parameters | ApacheAnt | 0.25 | 4.35 | 0.73 | 1.02 | **0.37** | 0.85 | 0.31 | 0.38 | 0.00 | 1.09 | 1.19 |
| remove assignment to parameters | ArgoUML | 1.52 | **0.29** | 1.24 | **0.34** | **1.88** | 0.82 | **0.00** | **1.12** | 0.65 | **0.22** | **0.25** |
| remove assignment to parameters | Xerces | 1.73 | **0.34** | - | 1.10 | **1.29** | 0.99 | **1.01** | 1.26 | **0.85** | **0.47** | **0.58** |
| remove control flag | ApacheAnt | **0.23** | **5.47** | **0.32** | **0.13** | **1.06** | **0.16** | 0.24 | 0.26 | 0.69 | **1.59** | **0.46** |
| remove control flag | ArgoUML | 1.38 | 2.19 | 1.47 | **0.97** | **1.65** | **1.06** | **0.10** | 1.53 | 0.37 | **0.69** | **0.22** |
| remove control flag | Xerces | 2.32 | 1.68 | - | 0.85 | 0.66 | 1.15 | **0.75** | 0.92 | **0.91** | 0.37 | **0.39** |
| remove parameter | ApacheAnt | **2.26** | **0.77** | 0.51 | 1.28 | **1.08** | **0.51** | **1.13** | **0.55** | 0.59 | 0.80 | 0.22 |
| remove parameter | ArgoUML | 1.10 | **0.93** | 1.16 | **1.12** | 1.42 | **0.95** | 0.06 | 0.88 | 0.89 | 0.36 | **0.59** |
| remove parameter | Xerces | 0.96 | 1.41 | - | 1.12 | 1.06 | **0.87** | **0.68** | 1.19 | 0.99 | **0.36** | 0.35 |
| rename method | ApacheAnt | 10.76 | **0.00** | **3.08** | 1.73 | **9.88** | **0.13** | **0.95** | **0.07** | **0.08** | **1.03** | **0.03** |
| rename method | ArgoUML | 1.29 | **0.90** | **1.01** | **1.13** | **0.98** | **1.22** | **0.12** | **1.10** | **0.96** | **0.34** | **0.26** |
| rename method | Xerces | 0.62 | 8.61 | - | 1.05 | 0.35 | 0.83 | 0.64 | 1.27 | 1.30 | 0.77 | 0.09 |
| replace data with object | ArgoUML | **0.38** | **0.95** | 2.38 | 1.19 | 1.20 | **1.57** | 0.12 | 1.02 | **0.19** | **0.01** | 0.39 |
| replace data with object | Xerces | **1.81** | 1.47 | - | **0.96** | **0.36** | 1.32 | **1.40** | **1.26** | **1.31** | **0.24** | 0.15 |
| replace exception with test | Xerces | **8.43** | **0.74** | - | **0.00** | **0.48** | 0.03 | **1.86** | 7.48 | 0.00 | 0.09 | 0.42 |
| replace magic number with constant | ApacheAnt | **0.37** | 10.04 | 0.86 | 0.55 | 0.51 | 1.15 | 0.57 | 0.02 | 0.26 | **0.81** | 0.57 |
| replace magic number with constant | ArgoUML | **1.95** | 0.04 | 1.87 | 0.28 | **2.18** | 0.56 | **0.76** | 0.77 | 0.57 | **0.12** | **0.26** |
| replace magic number with constant | Xerces | 0.77 | 2.64 | - | **0.92** | 0.61 | **3.63** | **1.03** | **1.05** | 0.90 | **0.54** | **0.51** |
| replace method with method object | ApacheAnt | **0.61** | **5.84** | **0.39** | 0.12 | **4.51** | **0.27** | **1.05** | **0.45** | **1.08** | **0.31** | **0.40** |
| replace method with method object | ArgoUML | 1.21 | 0.80 | 2.16 | 0.97 | **1.17** | 1.68 | 0.10 | **0.74** | **0.91** | **0.77** | **0.68** |
| replace method with method object | Xerces | 1.69 | 0.92 | - | 1.04 | 0.80 | **1.15** | 1.08 | 0.98 | 1.04 | **0.47** | 0.24 |
| replace nested cond guard clauses | ApacheAnt | **0.22** | **8.92** | **1.31** | 0.42 | **1.11** | **0.87** | 0.09 | 0.93 | **0.62** | **0.61** | **1.26** |
| replace nested cond guard clauses | ArgoUML | **0.66** | 3.86 | **0.44** | 0.98 | 2.50 | 0.87 | 0.01 | 1.48 | 0.86 | **1.03** | **0.55** |
| replace nested cond guard clauses | Xerces | **1.64** | 1.56 | - | 1.09 | 0.94 | **1.42** | 0.87 | 0.96 | 1.06 | 0.76 | **0.19** |
| separate query from modifier | Xerces | 0.76 | 3.31 | - | 1.04 | 0.42 | 1.63 | 0.60 | 1.33 | 0.54 | **3.56** | 0.95 |

*extract method* refactoring (see Table 7).

Concerning DIT, the metric measuring the depth of a class as the number of its ancestor classes, we expect strong ORs for refactoring operations dealing with changes applied to the class hierarchy (i.e., *push down method*, *pull up method*, *pull up field*, *push down field*, *form template method*, and *extract superclass*). However, we do not observe any statistically significant OR higher than one.

As for the NOC metric, counting the number of direct descendants (subclasses) of a class, we expected high ORs for refactoring operations acting on the class hierarchy. However, the ORs we found are either not statistically significant, or very close to 1 (see Table 7).

NOA (the number of operations added by a subclass) and NOO (the number of operations overridden by a subclass) are also related to class hierarchies and, in such cases, results confirm the conjecture that such metrics can relate with refactorings. Firstly, both metrics show high ORs with the *form template method* refactoring, which is often applied when in two subclasses there are very similar methods. These two methods are generally merged into a single one that is pulled up in the class hierarchy. For this reason, NOA and NOO also exhibit very high ORs with the *pull up method* and *pull up field* refactorings, even if these are not statistically significant.

RFC measures the coupling of a class and thus, we expect it to obtain high ORs for refactoring operations allowing a coupling reduction (e.g., inline method, move method, move field). Concerning the *inline method* refactoring, applied to merge two very coupled methods, we

Table 8: Quality metrics model: summary of results.

| Metric | Refactoring operations related to the metric: Expected | Found | Overlap |
|---|---|---|---|
| LOC | All | add parameter; extract superclass; form template method; inline temp move field; move method; pull up field; pull up method; push down field; push down method; replace exception with test | 39% |
| WMC | add parameter; consolidate cond expression; consolidate duplicate cond fragments; extract method; remove control flag; replace nested cond guard clauses | consolidate cond expression; consolidate duplicate cond fragments; remove control flag; replace nested cond guard clauses | 67% |
| DIT | pull up method; pull up field; push down field; push down method; form template method; extract superclass | introduce null object | 0% |
| NOC | pull up method; pull up field; push down field; push down method; form template method; extract superclass | add parameter; extract superclass; consolidate duplicate cond fragments; introduce explaining variable; pull up field; remove parameter | 20% |
| RFC | inline method; move field; move method | extract method; inline method; remove parameter | 20% |
| CBO | inline method; move field; move method; pull up method; pull up field; push down field; push down method; form template method | introduce null object; pull up field; push down method; replace data with object | 20% |
| LCOM | move field; move method | replace exception with test | 0% |
| NOA | pull up method; pull up field; push down field; push down method; form template method; extract superclass | form template method | 17% |
| NOO | pull up method; pull up field; push down field; push down method; form template method; extract superclass | form template method; push down field | 30% |
| CCBC | inline method; move field; move method; pull up method; pull up field; push down field; push down method; form template method; rename method | separate query from modifier | 0% |
| C3 | move field; move method; rename method | push down field; push down method | 0% |

Table 9: Code smells identified in each system (among all analyzed versions).

| System | Blob | CDSBP | Complex Class | Lazy Class | Long Method | LPL | Message Chain | Refused Bequest | Spaghetti Code | Speculative Generality | Feature Envy |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ApacheAnt | 85 | 370 | 0 | 167 | 110 | 12 | 0 | 5 | 9 | 40 | 62 |
| ArgoUML | 196 | 343 | 67 | 351 | 151 | 31 | 0 | 56 | 28 | 185 | 291 |
| Xerces | 328 | 792 | 48 | 664 | 700 | 17 | 0 | 852 | 71 | 124 | 34 |

found ORs higher than 1 for all object systems, showing that highly coupled classes have a higher chance of being involved in such refactoring. However, for operations like *move method* and *move field*, we found contradicting results. Specifically, for these two refactorings we found very high ORs on ApacheAnt (7.13 for *move method* and 6.82 for *move field*) together with ORs lower than 1 on the other two systems. We also found very high ORs for other refactoring operations that, however, do not allow to reduce coupling (see e.g., *rename method* with an OR of 9.88 in ApacheAnt).

CBO, also related to coupling, mainly exhibits high ORs for refactoring operations that are not related to a coupling reduction (e.g., *replace method with method object* with an OR of 3.63 in Xerces). The only expected result we found is that classes having high CBO (and thus, having several dependencies with other classes) have a higher chance of being involved in a *push down method* refactoring (OR equals 3.00) and generally have a higher chance of being involved in all refactoring operations moving code components among the class hierarchy. This result is expected since classes having a high CBO are also more likely to have inheritance dependencies with other classes. In fact, the CBO counts the number of objects with which a class has dependencies, including inheritances.

The structural cohesion metric LCOM does not provide any interesting result, generally showing low OR for the different refactoring operations. Some interesting results were achieved for the semantic cohesion metric C3, for which we observed an OR higher than 1 for *move method* and *move field* refactoring on ArgoUML. This indicates that some responsibilities of classes having low C3 (conceptual cohesion) are extracted from such classes. Finally, concerning the semantic coupling metric CCBC, it shows a high OR for the *separate query from modifier* refactoring. However, this refactoring operation does not deal with coupling reduction. While in some cases ORs higher than 1 are obtained for refactoring reducing coupling (e.g., *move method* on ApacheAnt), as already observed for the structural coupling metric RFC, this result is not confirmed on all the other systems, exhibiting ORs lower than 1.

Table 8 summarizes the results achieved for the quality metrics model by reporting for each of the investigated metrics:

1. The refactoring operations for which we expected some form of correlation. For example, we expect that classes having a high WMC value (WMC measures the code complexity) are more subject to refactoring operations aiming at reducing code complexity like, for example, *extract method*.

Table 10: Code smell model: OR of smells when building logistic regression model. Statistically significant ORs are reported in **bold face**.

| Refactoring | System | Blob | CDSBP | Complex Class | Lazy Class | Long Method | LPL | Refused Bequest | Spaghetti Code | Speculative Generality | Feature Envy |
|---|---|---|---|---|---|---|---|---|---|---|---|
| add parameter | ApacheAnt | **6.53** | 1.74 | 0.00 | 0.00 | **3.25** | 0.00 | 0.00 | **7.40** | 0.00 | 0.00 |
| add parameter | ArgoUML | **2.66** | **1.83** | 0.00 | **0.12** | **3.62** | 0.00 | 0.85 | 0.00 | 0.86 | 1.02 |
| add parameter | Xerces | 1.10 | **0.70** | 0.39 | 0.00 | **2.27** | **2.70** | **0.11** | **2.35** | **4.14** | 0.75 |
| consolidate cond expression | ApacheAnt | **9.33** | 0.46 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 195.80 | 0.00 | 0.00 |
| consolidate cond expression | ArgoUML | 3.29 | 0.90 | 0.00 | 0.00 | **5.16** | 0.00 | 0.00 | 0.00 | 0.00 | **3.71** |
| consolidate cond expression | Xerces | **2.45** | 0.91 | 0.00 | 0.00 | **1.79** | **5.44** | 0.37 | 0.61 | 1.98 | 0.00 |
| consolidate duplicate cond fragments | ApacheAnt | **2.72** | 1.55 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.04 | 3.96 | 0.00 |
| consolidate duplicate cond fragments | ArgoUML | 2.34 | **2.66** | 0.00 | 0.00 | **3.84** | 0.00 | 0.00 | 0.00 | 0.00 | **2.74** |
| consolidate duplicate cond fragments | Xerces | **1.40** | 1.08 | 1.84 | 0.00 | **5.33** | **4.49** | 0.90 | **1.97** | **1.80** | **3.44** |
| extract method | ApacheAnt | **2.76** | 1.83 | 0.00 | 0.00 | **4.02** | 0.00 | 0.00 | 0.57 | 0.00 | 0.00 |
| extract method | ArgoUML | **12.54** | 0.21 | 0.00 | 0.00 | **9.17** | 0.00 | 0.00 | 0.00 | 0.00 | 0.90 |
| extract method | Xerces | **1.56** | **1.88** | 0.43 | 0.00 | **5.94** | 0.00 | 0.00 | **4.56** | 0.93 | **3.47** |
| extract superclass | ArgoUML | 0.00 | 3.12 | 4.14 | 0.00 | 0.00 | 0.00 | 0.00 | 4.24 | 0.00 | 0.00 |
| form template method | ArgoUML | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| inline method | ApacheAnt | 2.43 | 0.84 | 0.00 | 0.00 | **45.79** | 0.00 | 0.00 | 1.14 | 0.00 | 0.00 |
| inline method | ArgoUML | 0.00 | 1.87 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 85.92 | 0.00 | 1.35 |
| inline method | Xerces | **3.29** | 1.62 | 0.00 | 0.00 | **3.23** | 0.00 | 0.00 | 0.00 | 0.22 | **5.41** |
| inline temp | ApacheAnt | **8.80** | **2.89** | 0.00 | 0.00 | **11.63** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| inline temp | ArgoUML | 1.80 | 2.02 | 0.00 | 0.00 | 2.71 | 0.00 | 0.00 | 0.00 | 0.00 | 0.83 |
| inline temp | Xerces | **2.28** | 1.41 | 1.13 | 0.00 | **5.02** | 2.10 | 0.00 | 0.00 | **2.18** | 0.00 |
| introduce assertion | ArgoUML | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.30 | 0.00 | 7.40 |
| introduce explaining variable | ApacheAnt | **5.15** | **2.66** | 0.00 | 0.00 | **6.88** | 4.75 | 0.00 | **4.69** | 4.48 | 0.00 |
| introduce explaining variable | ArgoUML | 1.56 | 0.78 | 0.00 | 0.00 | **5.06** | 0.00 | 0.00 | 0.00 | 0.00 | **2.14** |
| introduce explaining variable | Xerces | 1.48 | **2.11** | 0.00 | 0.00 | **3.73** | **8.17** | 0.24 | 0.98 | 1.61 | 2.39 |
| introduce null object | ArgoUML | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| introduce parameter object | Xerces | **14.14** | 0.00 | 0.00 | 0.00 | 0.48 | 0.00 | 0.00 | 48.74 | 0.00 | 0.00 |
| move field | ApacheAnt | 2.49 | 1.84 | 0.00 | 0.00 | **43.74** | 0.00 | 0.00 | 1.82 | 0.00 | 0.00 |
| move field | ArgoUML | 0.00 | 1.64 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 22.15 | 0.00 | 8.03 |
| move field | Xerces | **1.65** | **0.66** | 0.00 | **0.53** | **1.27** | 0.00 | 0.00 | 0.00 | **2.04** | 0.00 |
| move method | ApacheAnt | 1.43 | 0.49 | 0.00 | 0.00 | 2.92 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| move method | ArgoUML | 0.00 | 1.71 | 22.35 | 0.00 | 0.61 | 0.00 | 0.00 | 0.00 | 1.13 | 0.44 |
| move method | Xerces | **2.46** | **0.05** | 0.00 | 0.00 | 1.10 | 0.00 | 0.00 | 0.00 | **0.27** | 0.00 |
| pull up field | Xerces | 0.00 | 3.55 | 19.27 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| pull up method | Xerces | **11.95** | 0.00 | 17.86 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| push down field | Xerces | **16.43** | 0.00 | 0.00 | 0.00 | **0.21** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| push down method | Xerces | **26.79** | 0.00 | 16.28 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| remove assignment to parameters | ApacheAnt | **3.27** | 1.71 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| remove assignment to parameters | ArgoUML | 2.36 | 0.96 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.36 |
| remove assignment to parameters | Xerces | 0.92 | 0.89 | **8.79** | 0.00 | **2.00** | 6.47 | 0.00 | 0.00 | **2.82** | 0.00 |
| remove control flag | ApacheAnt | **8.13** | 0.67 | 0.00 | 0.00 | **5.69** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| remove control flag | ArgoUML | 0.40 | 0.87 | **0.08** | 0.00 | **26.12** | 0.00 | 0.00 | 0.00 | 0.54 | **9.35** |
| remove control flag | Xerces | **1.82** | 0.85 | **3.18** | 0.00 | **2.85** | 0.00 | 0.53 | 2.24 | **2.29** | 0.00 |
| remove parameter | ApacheAnt | **6.54** | **2.17** | 0.00 | 0.00 | **5.76** | 0.00 | 0.00 | **3.19** | 0.00 | 0.00 |
| remove parameter | ArgoUML | **3.28** | **2.38** | 0.00 | **0.14** | **3.85** | 0.00 | 0.99 | 0.00 | 1.23 | 0.66 |
| remove parameter | Xerces | 1.16 | 0.97 | 0.45 | 0.00 | **2.76** | 1.38 | **0.12** | **2.69** | 1.38 | 1.52 |
| rename method | ApacheAnt | **2.73** | **2.29** | 0.00 | 0.00 | **76.36** | 0.00 | 0.00 | 1.36 | 0.00 | 0.00 |
| rename method | ArgoUML | 0.00 | 1.21 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 189.30 | 2.37 | 0.54 |
| rename method | Xerces | **14.05** | 0.91 | 0.00 | 0.00 | **1.68** | 0.91 | **0.10** | 0.00 | **0.07** | **0.39** |
| replace data with object | ArgoUML | 0.00 | 4.16 | 21.65 | 0.00 | 0.00 | 0.00 | 0.00 | 43.26 | 0.00 | 0.00 |
| replace data with object | Xerces | **2.95** | 1.02 | 0.00 | 0.00 | 1.14 | 0.00 | 0.00 | 0.00 | **5.07** | 0.00 |
| replace exception with test | Xerces | 0.74 | 0.00 | 0.00 | 0.00 | 0.77 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| replace magic number with constant | ApacheAnt | 1.14 | **3.59** | 0.00 | 0.00 | 1.01 | 3.41 | 0.00 | 2.12 | 1.00 | 0.00 |
| replace magic number with constant | ArgoUML | **4.63** | **17.43** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.27 | 0.00 |
| replace magic number with constant | Xerces | **1.31** | **2.08** | 0.00 | 0.00 | **2.36** | 0.00 | **0.12** | 0.00 | **3.58** | 0.45 |
| replace method with method object | ApacheAnt | **16.46** | **4.43** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 13.70 | 0.00 | 0.00 |
| replace method with method object | ArgoUML | 0.44 | **1.79** | 0.00 | 0.00 | **3.90** | 0.00 | 1.11 | 0.00 | 0.00 | 1.14 |
| replace method with method object | Xerces | **3.41** | 0.80 | 0.41 | 0.00 | **1.53** | 0.00 | 0.00 | **6.71** | **3.92** | 1.05 |
| replace nested cond guard clauses | ApacheAnt | 3.09 | 0.84 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.81 | 0.00 | 0.00 |
| replace nested cond guard clauses | ArgoUML | 0.00 | 1.48 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.40 | 0.00 | **6.52** |
| replace nested cond guard clauses | Xerces | 1.06 | 0.99 | 0.00 | 0.00 | **11.34** | 0.00 | 0.46 | **3.97** | **2.75** | 0.59 |
| separate query from modifier | Xerces | **5.94** | 0.00 | 0.00 | 0.00 | **3.82** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

2. The refactoring operations for which we observed evidence of a relationship with quality metrics profile. In this case we mean refactoring operations for which we observed (i) a statistically significant OR higher than one for at least one of the object systems and (ii) consistent results (i.e., OR higher than one, even if not statistically significant) on the other systems.
3. The percentage of overlap between the set of expected refactorings (point 1) and the set of refactorings for which we actually observed some form of correlation (point 2).

The analysis of Table 8 highlights that *with very few exceptions, quality metrics do not show a clear relationship with refactoring*. The only exception is represented by the WMC metric, that seems to be able to indicate classes attracting the developers' refactoring attentions. As for the other metrics, none of them showed with strong evidence relation with refactoring. Particularly surprising are the results achieved with cohesion and coupling metrics, generally considered good indicators of source code components in need of refactoring [5]. It is important to point out that we are not claiming the opposite being generally true, but just reporting that *refactoring operations do not target classes exhibiting low cohesion and/or high coupling as much as expected.*

### 3.2. *To what extent are refactoring operations (i) executed on classes exhibiting code smells and (ii) able to remove code smells?*

Table 9 reports the number of classes affected by the different code smells we identified in the analyzed releases. Note that, for each system, we report the overall number of code smells identified across all the analyzed releases. This means that if a class is affected by a code smell in all the 33 analyzed Xerces releases, this class has been

Table 11: Number of refactorings performed on each type of code smell.

| Refactoring | System | Blob | CDSBP | Complex Class | Lazy Class | Long Method | LPL | Refused Bequest | Spaghetti Code | Speculative Generality | Feature Envy |
|---|---|---|---|---|---|---|---|---|---|---|---|
| add parameter | ApacheAnt | 20 | 18 | 0 | 0 | 33 | 0 | 0 | 24 | 0 | 0 |
| add parameter | ArgoUML | 15 | 23 | 0 | 1 | 13 | 0 | 1 | 0 | 4 | 19 |
| add parameter | Xerces | 131 | 63 | 3 | 0 | 201 | 5 | 3 | 18 | 71 | 5 |
| consolidate cond expression | ApacheAnt | 5 | 2 | 0 | 0 | 3 | 0 | 0 | 3 | 0 | 0 |
| consolidate cond expression | ArgoUML | 2 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 6 |
| consolidate cond expression | Xerces | 46 | 15 | 0 | 0 | 44 | 3 | 2 | 1 | 10 | 0 |
| consolidate duplicate cond fragments | ApacheAnt | 6 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| consolidate duplicate cond fragments | ArgoUML | 3 | 6 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 9 |
| consolidate duplicate cond fragments | Xerces | 125 | 43 | 10 | 0 | 193 | 6 | 9 | 12 | 33 | 15 |
| extract method | ApacheAnt | 7 | 10 | 0 | 0 | 9 | 0 | 0 | 2 | 0 | 0 |
| extract method | ArgoUML | 16 | 1 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 6 |
| extract method | Xerces | 53 | 24 | 1 | 0 | 84 | 0 | 0 | 11 | 10 | 7 |
| extract superclass | ArgoUML | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| form template method | ArgoUML | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| inline method | ApacheAnt | 2 | 1 | 0 | 0 | 16 | 0 | 0 | 6 | 0 | 0 |
| inline method | ArgoUML | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| inline method | Xerces | 31 | 8 | 0 | 0 | 31 | 0 | 0 | 0 | 1 | 6 |
| inline temp | ApacheAnt | 15 | 13 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 |
| inline temp | ArgoUML | 2 | 5 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 3 |
| inline temp | Xerces | 37 | 15 | 2 | 0 | 47 | 1 | 0 | 0 | 12 | 0 |
| introduce assertion | ArgoUML | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| introduce explaining variable | ApacheAnt | 17 | 19 | 0 | 0 | 38 | 1 | 0 | 24 | 1 | 0 |
| introduce explaining variable | ArgoUML | 2 | 2 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 8 |
| introduce explaining variable | Xerces | 45 | 29 | 0 | 0 | 63 | 4 | 1 | 2 | 14 | 4 |
| introduce null object | ArgoUML | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| introduce parameter object | Xerces | 8 | 0 | 0 | 0 | 4 | 0 | 0 | 3 | 0 | 0 |
| move field | ApacheAnt | 5 | 5 | 0 | 0 | 43 | 0 | 0 | 20 | 0 | 0 |
| move field | ArgoUML | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| move field | Xerces | 53 | 4 | 0 | 0 | 42 | 0 | 0 | 0 | 5 | 0 |
| move method | ApacheAnt | 1 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 |
| move method | ArgoUML | 0 | 15 | 3 | 0 | 4 | 0 | 0 | 0 | 2 | 4 |
| move method | Xerces | 71 | 3 | 0 | 0 | 62 | 0 | 0 | 0 | 3 | 0 |
| pull up field | Xerces | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pull up method | Xerces | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| push down field | Xerces | 9 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 |
| push down method | Xerces | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| remove assignment to parameters | ApacheAnt | 5 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| remove assignment to parameters | ArgoUML | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| remove assignment to parameters | Xerces | 12 | 6 | 4 | 0 | 18 | 1 | 0 | 0 | 6 | 0 |
| remove control flag | ApacheAnt | 5 | 2 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
| remove control flag | ArgoUML | 1 | 3 | 1 | 0 | 21 | 0 | 0 | 0 | 0 | 44 |
| remove control flag | Xerces | 26 | 4 | 0 | 0 | 42 | 0 | 0 | 0 | 5 | 0 |
| remove parameter | ApacheAnt | 20 | 19 | 0 | 0 | 29 | 0 | 0 | 16 | 0 | 0 |
| remove parameter | ArgoUML | 16 | 26 | 0 | 1 | 12 | 0 | 1 | 0 | 5 | 11 |
| remove parameter | Xerces | 91 | 42 | 2 | 0 | 140 | 2 | 2 | 15 | 22 | 7 |
| rename method | ApacheAnt | 13 | 14 | 0 | 0 | 112 | 0 | 0 | 44 | 0 | 0 |
| rename method | ArgoUML | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 5 |
| rename method | Xerces | 563 | 62 | 0 | 0 | 339 | 6 | 2 | 0 | 4 | 9 |
| replace data with object | ArgoUML | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| replace data with object | Xerces | 11 | 5 | 0 | 0 | 9 | 0 | 0 | 0 | 5 | 0 |
| replace exception with test | Xerces | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| replace magic number with constant | ApacheAnt | 17 | 62 | 0 | 0 | 16 | 1 | 0 | 8 | 1 | 0 |
| replace magic number with constant | ArgoUML | 9 | 46 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| replace magic number with constant | Xerces | 104 | 101 | 0 | 0 | 144 | 0 | 2 | 0 | 64 | 2 |
| replace method with method object | ApacheAnt | 17 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| replace method with method object | ArgoUML | 2 | 16 | 0 | 0 | 10 | 0 | 1 | 0 | 0 | 15 |
| replace method with method object | Xerces | 59 | 17 | 1 | 0 | 55 | 0 | 0 | 9 | 18 | 2 |
| replace nested cond guard clauses | ApacheAnt | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| replace nested cond guard clauses | ArgoUML | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| replace nested cond guard clauses | Xerces | 37 | 16 | 0 | 0 | 79 | 0 | 1 | 10 | 17 | 1 |
| separate query from modifier | Xerces | 10 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 |

counted 33 times. We did not find any Message Chain code smell. Thus, we will not discuss it in the following results analysis.

Table 10 reports the ORs obtained for the considered code smells when building a logistic regression model for data concerning each refactoring operation (as explained in Section 2). Moreover, we also show in Table 11, the number of refactorings performed on each type of code smell, and in Table 12 the percentage of code smells removed when developers performed refactoring actions.

The analysis of ORs reported in Table 10 highlights that Blob classes are generally subject to refactoring. A Blob is a large class implementing different responsibilities and centralizing most of the system behavior. Note that this is somewhat an expected result, and consistent with the findings related to the metric model (Table 8). Indeed, Blob classes are quite large in terms of LOCs and, as observed while discussing the quality metrics results, larger classes generally have a higher chance of being involved in a refactoring operation. This result is also confirmed by the fact that developers of the three object systems performed a total of 1,753 refactoring operations on classes affected by the Blob code smell (see Table 11). However, the data in Table 12 shows that the refactoring operations that actually removed the Blob code smell are mainly two: *move method* and *move field*. Specifically, in Xerces (the only system for which we have a good number of *move method* and *move field* refactoring operations performed on Blob classes), *move method* refactoring removes the Blob code smell in 71% of cases while *move field* refactoring in 30% of cases. By performing a manual analysis of such cases, we discovered as often a set of *move method* refactorings is performed to completely remove a responsibility from the Blob class and, in some cases, *move method* and *move field* refactorings are performed together as *extract class* refactoring (this type of refactoring is not detected by

Ref-Finder). For example, the class `XSchemaValidator` from the Xerces system has been refactored by the developers between releases 1.0.0 and 1.0.4. `XSchemaValidator` was composed of 100 methods and 74 attributes and, as stated in its comment, was an "*experimental implementation of a validator for the W3C schema language*". Developers removed this Blob class from the system by splitting its responsibilities across three new classes extracted from it in release 1.0.4 (i.e., `Schema`, `SchemaImporter`, and `SchemaParser`). This was done by (i) partially rewriting the code present in class `XSchemaValidator`, and (ii) by performing 52 *move field* and 31 *move method* operations from `XSchemaValidator` to the three new extracted classes.

Thus, while a Blob class generally represents a catalyst of several refactoring operations due to its size (i.e., high LOCs), *move method* and *move field* refactorings (or in combination as *extract class*) seem to be the only refactoring operations effective in removing this design problem from the system.

Classes affected by the Class Data Should Be Private (CDSBP) code smell also attracted several refactoring operations. However, it is worth noting that this is mainly due to the fact that this is the most diffused code smell we found (see Table 9). In fact, as shown in Table 12, no refactoring operations removed this code smell. The refactoring operation having this goal is the *encapsulate field*. However, we only found one instance of this refactoring in the ArgoUML system. What instead stands out from the analysis of the ORs reported in Table 10, is that classes affected by CDSBP have a much higher chance of being involved in *replace magic number with constant* refactoring operations (this chance is up to 17.43 higher). By manually analyzing those cases, we did not find a clear explanation for this phenomenon. However, two possible explanations are plausible from our point of view. The first is that developers are more prone to add new class fields (and thus to apply *replace magic number with constant* refactoring) in classes already containing fields (like those affected by the CDSBP code smell). The second is that the introduction of this code smell is favored by the application of the *replace magic number with constant* refactoring. Indeed, such refactoring implies the introduction of a new field within the class and it is possible that the added field is publicly exposed, introducing a CDSBP.

Particularly interesting are the results achieved for Complex and Lazy Classes. Both are poorly refactored by developers. On the one side, Lazy Classes are very simple classes, thus they should not create too much trouble during maintenance activities, and consequently developers are not particularly motivated to refactor them. For example, the interface `LayoutedObject` from ArgoUML reported in Listing 1 has never been refactored by ArgoUML developers until the last release considered in our study (0.34). Hence, this is an expected result. On the other side, the reason behind the very few refactorings performed on Complex Classes is likely their complexity. In

```
package org.argouml.uml.diagram.layout;

//This is the most common form of an layouted
//object.
public interface LayoutedObject {
}
```

Listing 1: Example of a Lazy Class never refactored by developers in ArgoUML

total, we observed just 27 refactoring operations on the 115 complex classes involved in our study (to be compared, as example, to the 1,753 performed on the 609 Blob classes). For example, the Complex Class `RegularExpression` from the Xerces system has never been refactored by the developers. By looking inside its source code we found that `RegularExpression` is a large class composed of 3,155 LOCs, and the 32 methods contained in it are very complex. To get an idea, these methods contain in total 126 `switch case` statements and 536 `if else` statements. Thus, refactoring this class would be very challenging for developers.

Conversely, classes containing Long Methods are widely refactored, for a total of 2,012 total refactorings. Firstly, it is interesting to note that 35% of classes affected by Long Methods are also Blobs and, as these latter, they also catalyze the refactoring attention of developers. In particular, classes affected by this code smell have:

- from 2.27 to 3.62 times more chances of being involved in an *add parameter* refactoring;
- from 4.02 to 9.17 times more chances of being involved in an *extract method* refactoring;
- form 3.23 to 45.79 times more chances of being involved in an *inline method* refactoring (no data for ArgoUML);
- from 3.73 to 6.88 times more chances of being involved in an *introducing explaining variable* refactoring;
- from 2.85 to 26.12 times more chances of being involved in a *remove control flag* refactoring;
- from 2.76 to 5.76 times more chances of being involved in a *remove parameter* refactoring;
- from 1.68 to 76.36 times more chances of being involved in a *rename method* refactoring (no data for ArgoUML).

However, as shown in Table 12, only some of these refactorings are applied by developers with the aim of removing the Long Method. The refactoring more often removing a Long Method is the *remove control flag* that helps in removing the code smell by reducing the method length. As expected, the other refactoring often removing the Long Method is the *extract method*, representing the most natural solution to this

Table 12: Perc. of smells removed by each refactoring. In **bold** values for which a refactoring has been applied at least 10 times on a smell.

| Refactoring | System | Blob | CDSBP | Complex Class | Lazy Class | Long Method | LPL | Refused Bequest | Spaghetti Code | Speculative Generality | Feature Envy |
|---|---|---|---|---|---|---|---|---|---|---|---|
| add parameter | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | **100%** | 0% | 0% |
| add parameter | ArgoUML | **0%** | **0%** | 0% | 0% | **15%** | 0% | 100% | 0% | 50% | **16%** |
| add parameter | Xerces | **4%** | **0%** | 0% | 0% | **6%** | 0% | 0% | **6%** | **7%** | 100% |
| consolidate cond expression | ApacheAnt | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 100% | 0% | 0% |
| consolidate cond expression | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 17% |
| consolidate cond expression | Xerces | **2%** | **0%** | 0% | 0% | **2%** | 0% | 0% | 0% | **0%** | 0% |
| consolidate duplicate cond fragments | ApacheAnt | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| consolidate duplicate cond fragments | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| consolidate duplicate cond fragments | Xerces | **2%** | **0%** | **0%** | 0% | **1%** | 0% | 0% | **25%** | **12%** | **93%** |
| extract method | ApacheAnt | 0% | **0%** | 0% | 0% | 22% | 0% | 0% | 100% | 0% | 0% |
| extract method | ArgoUML | **0%** | 0% | 0% | 0% | **40%** | 0% | 0% | 0% | 0% | 50% |
| extract method | Xerces | **0%** | **0%** | 0% | 0% | **11%** | 0% | 0% | **0%** | **0%** | 100% |
| extract superclass | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| form template method | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| inline method | ApacheAnt | 0% | 0% | 0% | 0% | **0%** | 0% | 0% | 100% | 0% | 0% |
| inline method | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| inline method | Xerces | **3%** | 0% | 0% | 0% | **3%** | 0% | 0% | 0% | 0% | 100% |
| inline temp | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | 0% | 0% | 0% |
| inline temp | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 33% |
| inline temp | Xerces | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | 0% | **0%** | 0% |
| introduce assertion | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 67% |
| introduce explaining variable | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | **100%** | 0% | 0% |
| introduce explaining variable | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 13% |
| introduce explaining variable | Xerces | **0%** | **0%** | 0% | 0% | **3%** | 0% | 0% | 0% | **0%** | 100% |
| introduce null object | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| introduce parameter object | Xerces | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| move field | ApacheAnt | 0% | 0% | 0% | 0% | **0%** | 0% | 0% | **100%** | 0% | 0% |
| move field | ArgoUML | 0% | **0%** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| move field | Xerces | **30%** | 0% | 0% | 0% | **0%** | 0% | 0% | 0% | 0% | 0% |
| move method | ApacheAnt | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| move method | ArgoUML | 0% | **0%** | 0% | 0% | 0% | 0% | 0% | 0% | 100% | 0% |
| move method | Xerces | **73%** | 0% | 0% | 0% | **0%** | 0% | 0% | 0% | 0% | 0% |
| pull up field | Xerces | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| pull up method | Xerces | 100% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| push down field | Xerces | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| push down method | Xerces | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| remove assignment to parameters | ApacheAnt | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| remove assignment to parameters | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 50% |
| remove assignment to parameters | Xerces | **0%** | 0% | 0% | 0% | **0%** | 0% | 0% | 0% | 0% | 0% |
| remove control flag | ApacheAnt | 0% | 0% | 0% | 0% | 25% | 0% | 0% | 0% | 0% | 0% |
| remove control flag | ArgoUML | 0% | 0% | 0% | 0% | **81%** | 0% | 0% | 0% | 0% | **9%** |
| remove control flag | Xerces | **4%** | 0% | 0% | 0% | **14%** | 0% | 0% | 0% | 0% | 0% |
| remove parameter | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | **100%** | 0% | 0% |
| remove parameter | ArgoUML | **0%** | **0%** | 0% | 0% | **7%** | 0% | 100% | 0% | 60% | **27%** |
| remove parameter | Xerces | **5%** | **0%** | 0% | 0% | **9%** | 0% | 0% | **0%** | **9%** | 100% |
| rename method | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | 100% | 0% | 0% |
| rename method | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 100% | 0% |
| rename method | Xerces | **9%** | **0%** | 0% | 0% | **57%** | 0% | 0% | 0% | 0% | 100% |
| replace data with object | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| replace data with object | Xerces | **7%** | 0% | 0% | 0% | 11% | 0% | 0% | 0% | 0% | 0% |
| replace exception with test | Xerces | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| replace magic number with constant | ApacheAnt | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | 100% | 0% | 0% |
| replace magic number with constant | ArgoUML | 0% | **0%** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| replace magic number with constant | Xerces | **0%** | **0%** | 0% | 0% | **4%** | 0% | 0% | 0% | **19%** | 100% |
| replace method with method object | ApacheAnt | **0%** | **0%** | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| replace method with method object | ArgoUML | 0% | **0%** | 0% | 0% | **30%** | 0% | 100% | 0% | 0% | **13%** |
| replace method with method object | Xerces | **5%** | **0%** | 0% | 0% | **4%** | 0% | 0% | 0% | **0%** | 100% |
| replace nested cond guard clauses | ApacheAnt | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| replace nested cond guard clauses | ArgoUML | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 17% |
| replace nested cond guard clauses | Xerces | **0%** | **0%** | 0% | 0% | **0%** | 0% | 0% | **0%** | **0%** | 100% |
| separate query from modifier | Xerces | **0%** | 0% | 0% | 0% | 22% | 0% | 0% | 0% | 0% | 0% |

code smell. This refactoring has been applied by Xerces developers between release 2.7.1 and release 2.8.0 on the Long Method `DOMSerializerImpl.writeToString(Node wnode)` to extract from it three new methods (i.e., `_getXmlVersion(Node node)`, `_getInputEncoding(Node node)`, `_getXmlEncoding(Node node)`) each one implementing a specific responsibility.

It can also be noted that the high number of *extract method* refactorings partially explains the high number of *rename method* refactorings performed on long methods. Indeed, the method undergoing an *extract method* refactoring is generally also renamed to reflect its new purpose. As for the *add parameter* refactoring, it sometimes helps to remove a Long Method. This is due to the fact that computations previously performed inside the method to obtain a result $r$ are now required to the classes invoking the long method through the passing of $r$ as parameter.

As for the other refactorings previously mentioned (i.e., *inline method*, *introducing explaining variable*, *remove parameter*) they are massively performed on classes affected by Long Method mainly due to the long size of the involved code component.

The Long Parameter List (LPL) code smell is rarely refactored by developers (just 30 refactorings in total) as well as the Refused Bequest code smell (25 refactorings). Classes affected by the Spaghetti Code code smell have a higher chance of being involved in an *add parameter* refactoring. This is a very expected result. In fact, these classes are generally composed by methods with few (or no) parameters. Note that, as shown in Table 12, this refactoring is able to remove the code smell in 100% of cases on ApacheAnt. However, a deeper analysis, reported in Table 12, reveals that also the *remove parameter* refactoring removes the Spaghetti Code code smell in 100% of cases on ApacheAnt. Our manual analysis revealed that the 16 *remove parameter* performed on Spaghetti Code in ApacheAnt were always executed together with an *add parameter* refactoring. In particular, the parameter was generally moved from methods having more than one parameter to methods having no parameters inside the same class.

For the Speculative Generality code smell, we did not observe any particular result, while it is interesting to note that in 93% of cases a *consolidate duplicate conditional fragments* refactoring operation is able to remove a Feature Envy code smell on Xerces (the only system on which we have data for this refactoring). This refactoring removes a fragment of code that is present in more than one branch of a conditional expression. This means that often, a high coupling between one method and the "envied class" (i.e., the class causing the Feature Envy in which the method should moved) is not really needed, but just emphasized by duplicated code.

In summary, *5,425 of the analyzed 12,922 refactoring operations are performed on code smells (42%)*. However, of these 5,425 *only 933 actually removed the code smell from the affected class (7% of total operations)* and 895 are attributable to only four code smells (i.e., Blob, Long Method, Spaghetti Code, and Feature Envy). Table 13 summarizes our findings for the studied code smells, highlighting for each of them (i) the refactoring operations for which we expected a correlation with the presence of code smells, (ii) the refactoring operations that we identified as applied on the code smell and able to often remove it, and (iii) the percentage overlap between the two previous explained sets. Looking at Table 13 we conclude that:

- *Only some of the analyzed code smells, such as Blob, Long Method, Spaghetti Code, and Feature Envy, increases the chances of the affected classes of being refactored.*

- *The effectiveness of refactoring operations in removing code smells is generally low. In the analyzed project releases, only 7% of the smells are removed through refactoring operations.*

## 4. Threats to Validity

This section discusses the threats that could affect the validity of our study. Threats to *construct validity* concern the relationship between theory and observation. The most important threat to construct validity to be discussed is how we assess source code quality in this paper. Specifically, we have chosen to use source code metrics, namely LOC, Chidamber & Kemerer metrics, conceptual cohesion and coupling. Clearly, there may be other metrics that may capture software quality, for example metrics computed by means of dynamic analysis. Nevertheless, as explained in Section 2.2, we have chosen a mix of metrics capturing source code size, structural and lexical characteristics. Another threat to validity concerns the identification of code smells. As explained in Section 2.2, we used a constraint-based approach to perform a preliminary detection of code smells (using low threshold values to avoid reducing the recall) followed by a manual analysis performed by two independent evaluators (with the aim of reducing imprecision and subjectiveness). Despite such process, we cannot exclude that some code smells were missed by our analysis or that false positives were considered. Finally, similar issues apply to the investigated refactorings, selected through a manual validation over an initial set detected by Ref-Finder. As pointed out by its authors [45], Ref-Finder has a very good recall (95%) while the precision is a bit lower (79%). However, in this study we back-up possible imprecisions by complementing Ref-Finder by manual validation.

Threats to *conclusion validity* concern the relationship between treatment and outcome. We use logistic regression models to identify correlations between metric values, and the presence of code smells with refactoring actions. Other than highlighting cases of significant correlations, we report and discuss OR values.

Threats to *internal validity* concern factors that could influence our observations. In particular, the fact that code smells disappear, may or may not be related to refactoring activities occurred between the observed releases. In other words, other changes could have produced such effects. However, although the performed analyses and the obtained results allow us to claim correlation and not causation, we corroborate our quantitative results by means of some qualitative analysis, aimed at illustrating examples in which specific kinds of refactorings helped to remove some code smells.

Threats to *external validity* concern the generalization of our findings. The study is limited to three Java projects, because we preferred to observe fewer projects over a long period of evolution history, rather than many projects for a short period. This better allowed us to observe refactorings, that often happen during specific periods of a project lifetime [22]. We considered open source systems for our analysis, since the source code of commercial ones are not available. However, we provided data and tools used for the investigation in order to allow a replication on different (both open source and commercial) systems. Last, but not least, as mentioned in Section 2, this choice to analyze few systems was also due to the need for manually validating refactorings and smells, rather than just relying on tool output. In any case, further studies are therefore needed to confirm (or refute) our results. Also, the findings obtained for the investigated code smells may or may not apply to other kinds of code smells, for example those—such as Divergent Change or Parallel Inheritance—that can be detected using change history metrics [43].

## 5. 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 Mens *et al.* [33] and Bavota *et al.* [6] for a complete survey on the most recent approaches). However, our paper is mostly

Table 13: Code Smell model: Summary of the achieved results

| Code Smell | Refactoring operations related to the metric | | Overlap |
|---|---|---|---|
| | Expected | Found | |
| Blob | extract class; move method; move field | extract class (as combination of move method and move field); move method; move field | 100% |
| CDSBP | encapsulate field | replace magic number with constant | 0% |
| Complex Class | extract method; consolidate conditional expression; move method; extract class | - | 0% |
| Lazy Class | inline class | - | 0% |
| Long Method | extract method; remove control flag; consolidate conditional expression | extract method; remove control flag; consolidate conditional expression; add parameter; remove parameter; inline method; introducing explaining variable; rename method | 38% |
| LPL | introduce parameter object | - | 0% |
| Refused Bequest | push down method; push down field; replace inheritance with delegation | - | 0% |
| Spaghetti Code | add parameter | add parameter; remove parameter | 50% |
| Speculative Generality | collapse hierarchy | - | 0% |
| Feature Envy | move method; extract method; consolidate duplicate conditional fragments | consolidate duplicate conditional fragments | 33% |

related to work analyzing how developers refactor source code. In the following, we discuss only some of the existing approaches that automatically support refactoring operations, while we provide a full overview of the related literature on the works analyzing how developers refactor source code.

### 5.1. Automated Refactoring: Methods and Tools

Different approaches have been defined to identify the better way in which to refactor the source code. O'Keeffe and O'Cinneide [40] formulate the task of refactoring as a search problem in the space of alternative designs. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics (i.e., the CK metrics [17]) that accurately reflects refactoring goals. Atkinson and King [2] present a low-cost, syntactic approach for automatically discovering opportunities for refactoring the source code. The proposed approach uses the symbol table and reference information together with simple code metrics, such as line and statement counts. Moreover, only structural metrics are used to guide refactoring. Maruyama and Shima [32] present a mechanism that automatically refactors methods in object-oriented frameworks in order to improve the reusability of frameworks. For this purpose, the authors use weighted dependence graphs, when the weight of the edges is based on the modification histories of the methods. Casais [16] proposes several algorithms to restructure class hierarchies to maximize abstraction, while Moore [35] proposes a method where existing classes with a low quality are replaced with a new set of classes where their methods are optimally factored aiming at minimizing code duplication.

A fully automated approach might be undesirable, as developers might want to have the last word on the refactoring activities to perform. Semi-automatic approaches requiring the human interaction have also been presented to support refactoring activities. Opdyke [41] developed the first tool providing semi-automatic refactoring support, which was implemented in the Refactoring Browser [47]. Simon *et al.* [51] provide a metric-based visualization tool to support the software engineer in the identification of source code components that needs refactoring. Bodhuin *et al.* [11] introduce SORMASA, SOftware Refactoring using software Metrics And Search Algorithms, a refactoring decision support tool based on optimization techniques, namely Genetic Algorithms. Almost all the proposed approaches use design metrics to guide refactoring. The relation between design metrics and refactoring has been analyzed by several authors. DuBois *et al.* [19] analyze how refactorings manipulate coupling and cohesion characteristics, and how to identify refactoring opportunities that improve these characteristics. They provide practical guidelines for the optimal usage of refactoring in a software maintenance process. Another interesting study is presented by Sahraoui *et al.* [48], that propose to investigate whether some object-oriented metrics can be used as indicators for automatically detecting situations where a particular transformation can be applied to improve the quality of a system. The detection process is based on the analysis of the impact of various transformations on these object-oriented metrics using quality estimation models. Tsantalis *et al.* [54] presented JDeodorant, a tool for detecting Feature Envy smells with the aim of suggesting move method refactoring opportunities. In particular, for each method of the system, their approach forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. In its current version JDeodorant[6] is also able to refactor code in order to remove three other code smells (i.e., State Checking, Long Method, and God Classes). Finally, it is worth mentioning the work by Bavota *et al.* [8], in which the authors applied Relational Topic Model (RTM) in order to find *Move Class* Refactoring opportunities.

### 5.2. Empirical Studies on Refactoring

Wang *et al.* [55] performed a survey with ten professional developers with the aim of identifying the major factors that motivate their refactoring activities. The author identified twelve different factors pushing developers

[6] `http://www.jdeodorant.com/`

to adopt refactoring practices and classified them in *intrinsic motivators* and *external motivators*. Intrinsic motivators are those for which developers do not obtain external rewards. An example of intrinsic motivators is *Responsibility with Code Authorship*, i.e., developers want to ensure high quality for their code. What we miss here, is a clear definition of high-quality code (e.g., as measured by quality metrics?). On the other side, an example of external motivators is *Recognitions from Others*, i.e., high technical ability can help the software developers gain recognitions. Note that, unlike our work, in the paper by Wang *et al.* [55] the relationship between code quality (e.g., presence of code smells, quality metrics, change-proneness) and classes refactored by developers is not analyzed.

Murphy-Hill *et al.* [38] analyzed eight different datasets trying to understand how developers perform refactoring. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment, data from the Eclipse Usage Collector aggregating activities of 13,000 developers for almost one year, and information extracted from versioning systems. Some of the several interesting findings they found were (i) programmers rarely (almost 10% of times) configure refactoring tools, (ii) commit messages do not help in predicting refactoring, since rarely developers explicitly report their refactoring activities in them, (iii) developers often interleave refactoring with other programming activities, and (iv) most of the refactoring operations (close to 90%) are manually performed by developers without the help of any tool. In the design of our empirical study we took into account one of these important conclusions: commit messages do not help in predicting refactoring. For this reason we detected refactorings using Ref-Finder, that performs detection through code analysis. Differently from our work, Emerson *et al.* [38] did not analyze the characteristics of code artifacts generally object of refactoring by developers.

Kim *et al.* [27] presented a survey performed with 328 Microsoft engineers (of which 83% developers) to understand (i) the participants own refactoring definition, (ii) when and how they refactor code, (iii) if refactoring tools are used by developers and (iv) developers perception toward the benefits, risks, and challenges of refactoring [27]. The main findings of this study were that:

- While developers recognize refactoring as a way to improve the quality of a software, in almost 50% of cases they do not define refactoring as a behavior-preserving operation.

- The most important symptom that pushes developers to perform refactoring is low readability of source code.

- 51% of developers manually perform refactoring.

- The main benefits the developers observed from the refactoring were improved readability (43%) and improved maintainability (30%).

- The main risk developers fear when performing refactoring operations is bug introduction (77%).

Kim *et al.* [27] also reported the results of a quantitative analysis performed on the Windows 7 change history showing that refactored modules experienced a higher reduction in the number of inter-module dependencies and post-release defects than other modules. Differently from the study of Kim *et al.* [27], our work quantitatively analyzes if developers focus their refactoring attentions on classes having a low quality, as indicated by quality metrics, and code smells.

Finally, a number of works have studied the relationship between refactoring and software quality. Bavota *et al.* [4] conducted a study aimed at investigating to what extent refactoring activities induce faults. They show that refactorings involving hierarchies (e.g., *pull down method*) induce faults very frequently. Conversely, other kinds of refactorings are likely to be harmless in practice. We share with this work the dataset of refactoring operations used to run our study.

Stroggylos and Spinellis [52] 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).

Szoke *et al.* [53] 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 [1] investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their findings highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in some other classes.

Moser *et al.* [36] conducted a case study in an industrial environment aimed at investigating the impact of refactoring on the productivity of an agile team and on the quality of the code they produce. The achieved results show that refactoring not only increases software quality but it also increases developers' productivity.

## 6. Conclusion

This paper reported an empirical study aimed at investigating the characteristics of code components increasing their changes of being subject to refactoring operations. In particular, we verified whether refactoring activities occur on classes for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactorings. The study has been conducted on 63 releases of three open source projects, and required the manual analysis of 15,008 refactoring operations and 5,478 smells.

Our results highlighted that, with very few exceptions, quality metrics do not show a clear relationship with refactoring. One possible interpretation of such a finding can be found in a survey we recently performed with developers about their perception about some code smells [42]. Indeed, on the one hand developers found that only particularly serious smells (in terms of metrics) are worthwhile of being refactored. On the other hand, they also pointed out that in some cases metrics may not be *per se* indicators of smells: for example, some classes—e.g., implementing parsers or complex algorithms—might intrinsically exhibit anomalous metric profiles, without necessarily being considered as refactoring opportunities.

Almost 40% of the analyzed refactorings has been performed on classes affected by smells. However, just 7% of them actually removed the smell. In other words, it is possible that the refactoring only *mitigated* the problem, without however necessarily removing completely the smell.

This work is mainly exploratory in nature, as it is aimed at empirically investigating a phenomenon—which characteristics of classes promote refactoring operations—from a **quantitative** point-of-view. Nevertheless, there are different possible uses one can make of the results of this paper. When *building* recommendation tools aimed at highlighting refactoring opportunities to developers it must be taken into account that, at least among the code characteristics considered in this paper—i.e., code metrics, presence of smells—there is no silver bullet able to indicate which code artifacts are in need of refactoring. Future work in this area should aim at learning something from the past refactorings made by developers, in order to suggest refactoring recommendations more suitable for them.

Also, when *evaluating* refactoring recommendation tools the developer's point-of-view cannot be ignored. Often such tools are just evaluated by verifying if the refactorings they recommend are able to improve some quality metric values and/or to remove smells. However, our study indicates that the developer's point-of-view of classes in need of refactoring does not always match with these "quality indicators".

## References


[1] Alshayeb, M. 2009. Empirical investigation of refactoring effect on software quality. *Information and Software Technology 51,* 9, 1319 – 1326.

[2] Atkinson, D. C. and King, T. 2005. Lightweight detection of program refactorings. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference.* IEEE CS Press, Taipei, Taiwan, 663–670.

[3] Basili, V. R., Briand, L., and Melo, W. L. 1995. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering 22,* 10, 751–761.

[4] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. 2012. When does a refactoring induce bugs? an empirical study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation.* SCAM '12. 104–113.

[5] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. 2013. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 1–48.

[6] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer Berlin Heidelberg, 387–419.

[7] Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. 2013. An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering.* ICSE '13. 692–701.

[8] Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. 2014. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Trans. Softw. Eng. Methodol. 23,* 1, 4.

[9] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Trans. Software Eng. 40,* 7, 671–694.

[10] Binkley, A. B. and Schach, S. R. 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th International Conference on Software Engineering.* Kyoto, Japan, 452–455.

[11] Bodhuin, T., Canfora, G., and Troiano, L. 2007. SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *Proceedings of 1st Workshop on Refactoring Tools.* Berlin, Germany, 23–24.

[12] Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., and Ben Chikha, S. 2013. Competitive coevolutionary code-smells detection. In *Search Based Software Engineering.* Lecture Notes in Computer Science Series, vol. 8084. Springer Berlin Heidelberg, 50–65.

[13] Briand, L. C., Wuest, J., and Lounis, H. 1999. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the 15th IEEE International Conference on Software Maintenance.* IEEE Press, Oxford, UK, 475–482.

[14] Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H. 1999. Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the 21st International Conference on Software Engineering.* ACM Press, Los Angeles, California, United States, 345–354.

[15] Brown, W. J., Malveau, R. C., Brown, W. H., McCormick III, H. W., and Mowbray, T. J. 1998. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis* 1st Ed. John Wiley and Sons.

[16] Casais, E. 1992. An incremental class reorganization approach. In *Proceedings of the 6th European Conference on Object-Oriented Programming.* Utrecht, the Netherlands, 114–132.

[17] Chidamber, S. R. and Kemerer, C. F. 1994. A metrics suite for object oriented design. *Transactions on Software Engineering 20,* 6, 476–493.

[18] Crawford, S., McIntosh, A., and Pregibon, D. 1985. An analysis of static metrics and faults in c software. *Journal of Systems and Software 5,* 1, 37 – 48.

[19] Du Bois, B., Demeyer, S., and Verelst, J. 2004. Refactoring - improving coupling and cohesion of existing code. In *Proceedings of 11th Working Conference on Reverse Engineering.* IEEE CS Press, Delft, the Netherlands, 144–151.

[20] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. 2011. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* ACM, 1037–1039.

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

[22] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. 1999b. *Refactoring: Improving the Design of Ex-*

*isting Code*. Addison-Wesley.

[23] Gyimóthy, T., Ferenc, R., and Siket, I. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering 31,* 10, 897–910.

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

[25] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *Software Maintenance, 2002. Proceedings. International Conference on*. 576 – 585.

[26] Kessentini, M., Vaucher, S., and Sahraoui, H. 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. ACM, 113–122.

[27] Kim, M., Zimmermann, T., and Nagappan, N. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*.

[28] Leitch, R. and Stroulia, E. 2003. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*. 309 – 322.

[29] Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimóthy, T., and Chrisochoides, N. 2009. Modelling class cohesion as mixtures of latent topics. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*. IEEE Press, Edmonton, Canada, 233–242.

[30] Lorenz, M. and Kidd, J. 1994. *Object-Oriented Software Metrics*. Prentice-Hall.

[31] Marcus, A., Poshyvanyk, D., and Ferenc, R. 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transaction on Software Engineering 34,* 2, 287–300.

[32] Maruyama, K. and Shima, K. 1999. Automatic method refactoring using weighted dependence graphs. In *Proceedings of 21st International Conference on Software Engineering*. ACM Press, Los Alamitos, California, USA, 236–245.

[33] Mens, T. and Tourwé, T. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering 30,* 2, 126–139.

[34] Moha, N., Guéhéneuc, Y., Duchien, L., and Le Meur, A. 2010. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering 36,* 1, 20–36.

[35] Moore, I. 1996. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, San Jose, California, USA, 235–250.

[36] Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., and Succi, G. 2008. Balancing agility and formalism in software engineering. Springer-Verlag, Berlin, Heidelberg, Chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, 252–266.

[37] Moser, R., Sillitti, A., Abrahamsson, P., and Succi, G. 2006. Does refactoring improve reusability? In *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components*. ICSR'06. Springer-Verlag, Berlin, Heidelberg, 287–297.

[38] Murphy-Hill, E., Parnin, C., and Black, A. P. 2011. How we refactor, and how we know it. *Transactions on Software Engineering 38,* 1, 5–18.

[39] O'Brien, R. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity 41,* 5, 673.

[40] O'Keeffe, M. and O'Cinneide, M. 2006. Search-based software maintenance. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Bari, Italy, 249–260.

[41] Opdyke, W. F. 1992. Refactoring object-oriented framework. Ph.D. thesis, University of Illinois.

[42] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. 101–110.

[43] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. 2015. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*.

[44] Poshyvanyk, D., Marcus, A., Ferenc, R., and Gyimóthy, T. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering 14,* 1, 5–32.

[45] Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. 2010. Template-based reconstruction of complex refactorings. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 1–10.

[46] Ratzinger, J., Fischer, M., and Gall, H. 2005. Improving evolvability through refactoring. In *Proceedings of the 2005 international workshop on Mining software repositories*. MSR '05. 1–5.

[47] Roberts, D., Brant, J., and Johnson, R. 1997. A refactoring tool for smalltalk. *Theory and Practice of Object Systems 3,* 4, 253–263.

[48] Sahraoui, H. A., Godin, R., and Miceli, T. 2000. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proceedings of the 16th International Conference on Software Maintenance*. IEEE CS Press, Washington, Columbia, USA, 154.

[49] Shatnawi, R. and Li, W. 2011. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications 5,* 4, 127–149.

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

[51] Simon, F., Steinbrückner, F., and Lewerentz, C. 2001. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Lisbon, Portugal, 30–38.

[52] Stroggylos, K. and Spinellis, D. 2007. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*. WoSQ '07. IEEE Computer Society, Washington, DC, USA, 10–.

[53] Szoke, G., Antal, G., Nagy, C., Ferenc, R., and Gyimóthy, T. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 95–104.

[54] Tsantalis, N. and Chatzigeorgiou, A. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering 35,* 3, 347–367.

[55] Wang, Y. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. 413 –416.