Why Developers Refactor Source Code:
A Mining-based Study

JEVGENIJA PANTIUCHINA, Università della Svizzera italiana
FIORELLA ZAMPETTI, University of Sannio
SIMONE SCALABRINO, VALENTINA PIANTADOSI, and ROCCO OLIVETO,
University of Molise
GABRIELE BAVOTA, Università della Svizzera italiana
MASSIMILIANO DI PENTA, University of Sannio

Refactoring aims at improving code non-functional attributes without modifying its external behavior. Previous studies investigated the motivations behind refactoring by surveying developers. With the aim of generalizing and complementing their findings, we present a large-scale study quantitatively and qualitatively investigating *why* developers perform refactoring in open source projects. First, we mine 287,813 refactoring operations performed in the history of 150 systems. Using this dataset, we investigate the interplay between refactoring operations and process (e.g., previous changes/fixes) and product (e.g., quality metrics) metrics. Then, we manually analyze 551 merged pull requests implementing refactoring operations and classify the motivations behind the implemented refactorings (e.g., removal of code duplication). Our results led to (i) quantitative evidence of the relationship existing between certain process/product metrics and refactoring operations and (ii) a detailed taxonomy, generalizing and complementing the ones existing in the literature, of motivations pushing developers to refactor source code.

CCS Concepts: • **Software and its engineering** → **Maintaining software**;

Additional Key Words and Phrases: Refactoring, empirical software engineering

ACM Reference format:
Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 29 (September 2020), 30 pages.
https://doi.org/10.1145/3408302

ACM Transactions on Software Engineering and Methodology, Vol. 29, No. 4, Article 29. Pub. date: September 2020.

29

1 INTRODUCTION

Software refactoring has been widely studied in the research community, with most of the works falling into three main research threads: (i) approaches aimed at identifying refactoring opportunities [83], (ii) techniques to recommend refactoring solutions for a given design flaw [36], and (iii) empirical studies looking at software refactoring from many different perspectives [43, 80, 91, 104, 105]. The knowledge of motivations pushing developers to perform refactoring [91] can help in building recommender systems able to propose suitable solutions for that. For this reason, understanding when and why developers perform refactoring has been the goal of many previous studies [39, 60, 91, 104, 105].

Some of these studies tried to answer this question by looking at specific factors that might correlate with refactoring operations, such as code quality proxies (i.e., quantitative measures providing indications about the internal quality of code components, such as quality metrics or code smells) [39, 104]. While valuable, these studies provide limited insights into the reasons behind the performed refactorings, since their analysis is mostly quantitative and limited to a small number of factors. Other studies opted for a more qualitative approach by interviewing developers [60, 105] to identify the major factors that motivate their refactorings. Although these studies have pioneered the investigation of the reasons pushing developers to refactor their code, as observed by Silva et al. [91], the previously mentioned surveys are general purpose, meaning that they do not ask developers to *justify* specific refactorings they performed, but rather study refactoring habits in general. To address this limitation, Silva et al. [91] interviewed developers who authored 222 refactoring-related commits to understand the reasons behind these specific operations.

Stemming from the studies discussed above and to generalize their findings [60, 91, 105], this article describes large-scale mining study combining quantitative and qualitative analyses to investigate the motivations behind refactoring operations, by observing code and discussions rather than interviewing developers. From a quantitative point of view, we mine the change history of 150 Java repositories hosted on GitHub to extract 287,813 refactoring operations of 25 different types performed by developers through the RMiner tool [99]. Then, we analyze product- (e.g., slopes indicating whether the quality of code components as assessed by quality metrics is decreasing over time) and process-related (e.g., source code change- and fault-proneness) factors that contribute to trigger refactoring actions. As compared to previous work [39, 104], we consider a more comprehensive set of factors and, more importantly, analyze them in a single model rather than in isolation, showing which ones are related to refactoring operations. From a qualitative point of view, we use the same set of systems to manually analyze a statistically significant sample of 551 pull requests (PRs) in which (i) developers discuss refactoring **and** (ii) RMiner identifies at least one refactoring operation. Through a manual analysis, we identify the rationale of the refactoring change, and whether it is the main intent of the change or, rather, they are triggered by the code review process of the PR. As main contribution of this analysis, we defined an extensive taxonomy of 67 motivations pushing developers to implement refactoring operations. Our qualitative analysis complements and generalizes the findings in previous survey-based studies [60, 91, 105] by investigating the same research question with a completely different experimental design.

As compared to the most similar work (i.e., Silva et al. [91]), the following notable differences can be highlighted for what concerns the study design and findings:

- *Study Design: surveying developers vs analyzing their activities.* While Silva et al. contacted the developers authoring the refactorings asking their motivations for the implemented changes, we manually inspect pull requests implementing refactorings by analyzing their discussion and related commits to create our taxonomy of motivations. Investigating the same research question with two different experimental designs can lead to additional insights and helps in generalizing previous findings.

- *Study Design: complementing qualitative and quantitative analysis.* In our work, we analyze the motivations behind refactoring operations not only from a qualitative perspective (as done by Silva et al. [91]) but also by quantitatively studying the influence of product and process metrics on the triggering of refactoring operations. Also, to the best of our knowledge, our study is the first one analyzing these metrics in a single model rather than in isolation (as done in Reference [39], for example).
- *Findings: complementing and generalizing Silva et al. [91].* As output of their study, Silva et al. defined a list of 44 motivations for 12 frequently applied refactoring operations. Our taxonomy, besides confirming 41 of their motivations, thus improving the generalizability of their findings, includes 26 additional ones that are not covered in the previous study.

Our quantitative analysis indicates that code readability and process-related factors correlate with the changes a commit containing refactoring operations has. As the main result of the qualitative analysis, we provide a comprehensive taxonomy of 67 categories of motivations leading developers to refactoring operations. We describe and exemplify each category, and discuss its implications in refactoring research and practice.

2 STUDY DESIGN

The *goal* of this study is to quantitatively and qualitatively analyze the context in which refactoring operations occur in open source projects, with the aim of identifying the circumstances that may make a refactoring happen. The *quality focus* relates not only to code quality but also to the improvement of the software development process. The *context* consists of 287,813 refactoring actions automatically identified in 150 open-source projects and, for the qualitative analysis, of 551 manually analyzed PRs mentioning refactoring operations and linked onto refactoring-related commits.

We address the following two research questions (RQs):

RQ$_1$: *Which product and process-related factors relate with an increase of refactoring operation chances?* We are interested in studying if various source code features or process features correlate with the presence of refactoring operations in a commit.

RQ$_2$: *What are the reasons for performing a refactoring operation?* We investigate the rationale behind refactoring opportunities. We consider refactorings occurred in PRs, and perform a qualitative analysis of developers' discussions over the PR. Also, since previous work found that most refactoring operations occur with other changes [80], by analyzing PRs we give a closer look at this phenomenon, investigating if the refactoring was tangled with other changes, and looking at whether the refactoring was the primary purpose of the PR. We decided to answer this RQ by looking at PRs rather than at commits implementing refactorings, since PRs offer a richer set of information to analyze to derive the rationale behind refactoring operations. Indeed, they often feature a discussion among developers that can help in better understanding what the goal of the implemented change was.

The formulated RQs investigate the same phenomenon (i.e., what the motivations for refactoring operations are) from two different perspectives (quantitative—RQ$_1$ vs. qualitative—RQ$_2$). The catalog of motivations identified in the two RQs can complement and support each other.

2.1 Study Context

We identified the projects to be studied among repositories hosted on GitHub. Since the infrastructure used in our study (e.g., the refactoring detection tool) only supports Java, we focus on Java projects. Among all Java projects on GitHub, we aim at studying active projects having
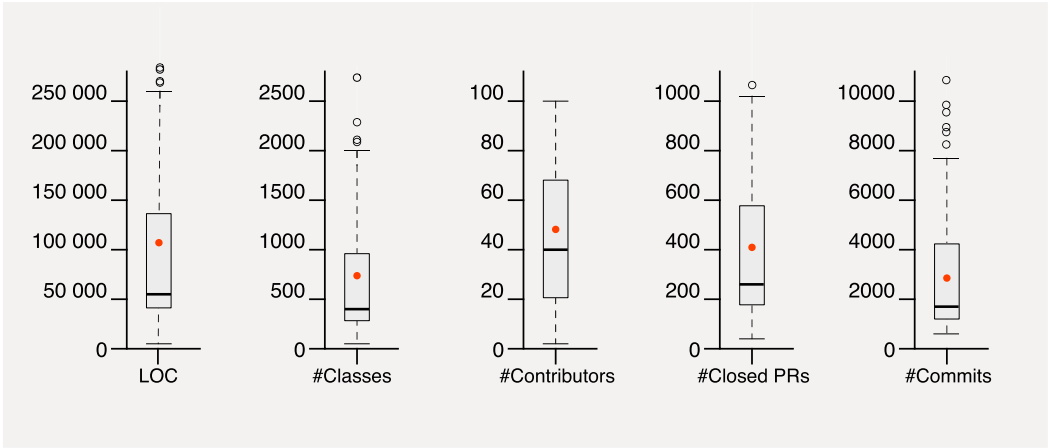
Fig. 1. Characteristics of the 150 projects used in our study.

a non-trivial change history to study (needed to mine the PRs needed for our study) and not representing personal and/or toy projects (e.g., a project created by a student during an assignment). To identify these projects we applied a number of selection criteria, only retaining projects having:

- *At least 5 contributors and 1 fork*, to exclude personal/toy projects.
- *At least 500 commits and 100 PRs*, to exclude projects having a short change history and unlikely to provide useful PRs for our study.
- *Modified at least once in the period Jan-May 2019*, to exclude inactive projects at the time in which this study has been run.

From the set of 303 remaining projects, we randomly selected 150 of them for our study (list available in Reference [27]). The choice of selecting a subset of the 303 projects was dictated by the computationally expensive data extraction process adopted in our study. Indeed, as detailed in the following, besides detecting refactoring operations, we computed 42 product- and process-metrics (e.g., code quality metrics, change-proneness of classes) for each of the 213,102 commits in the studied projects. This process took three months on a 56-core server. Figure 1 reports boxplots depicting the distribution of Lines of Code (LOC), number of classes (#Classes), number of contributors (#Contributors), number of closed PRs (#Closed PRs), and number of commits (#Commits) for the analyzed 150 systems. The raw data from which this figure has been created is available in our replication package [27].

We used the RMiner tool [99] to detect the refactorings implemented by developers in the studied projects. We focus on commits performed in the master/default branch of each project. We have chosen RMiner due to its high reported precision (98%) and recall (87%) [99]. RMiner takes as input two consecutive commits and provides as output the set of detected refactorings (see [99] for the supported refactorings).

2.2 Quantitative Analysis (RQ$_1$)

The occurrences of the detected refactorings constitute the *dependent variable* for RQ$_1$. As *independent variables*, we consider process-/product-related factors for each snapshot s_i (commit) of the master branch.

Table 1. Quality Metrics (Product-related Factors)

Metric	Description
CBO ✓	Coupling Between Object classes: measures the dependencies a class has [46]
WMC ✓	Weighted Methods per Class: sums the cyclomatic complexity of the methods in a class [46]
RFC ✓	Response For a Class: the number of methods in a class plus the number of remote methods that are called recursively through the entire call tree [46]
ELOC ✓	Effective Lines Of Code: the lines of code excluding blank lines and comments
NOM ✓	Number Of Methods in a class
NOPM ✓	Number Of Public Methods in a class
DIT ✓	Depth of Inheritance Tree: the length of the path from a class to its farthest ancestor [46]
NOC ✓	Number Of Children (direct subclasses) of a class
NOF ✓	Number Of Fields declared in a class
NOSF ✓	Number Of Static Fields declared in a class
NOPF	Number Of Public Fields declared in a class
NOSM ✓	Number Of Static Methods in a class
NOSI ✓	Number Of Static Invocations of a class
HsLCOM ✓	Henderson-Sellers revised Lack of Cohesion Of Methods (LCOM): a class cohesion metric based on the sharing of local instance variables by the methods of the class [46]. HsLCOM dresses limitations of the original LCOM [58]
C3 ✓	Conceptual Cohesion of Classes: avg. textual similarity between all pairs of methods in a class [74]
StrRead ✓	Structural readability: uses structural aspects (e.g., line length) to model code readability [41]
ComRead ✓	Comprehensive readability model: combines structural, visual (e.g., alignment) and textual features (e.g., comments readability) [90]

Near each factor, we indicate whether (✓) it was retained. The factors retained in the model are also highlighted in bold.

2.2.1 Identification of Product and Process Metrics. The considered metrics are summarized in Tables 1, 2, and 3 and described in the following. The selection of these metrics (detailed in the following) is based on the will to include in our study:

(1) *Metrics capturing code quality from different perspectives (Table 1).* We included both structural and semantic (i.e., textual) metrics that have been shown to capture orthogonal code quality aspects [74]. Also, we considered the recent readability metrics proposed in the literature [41, 90] that have been shown to highly correlate with the developers' assessment of code readability.

(2) *Code smells and quality issues widely studied in the literature (Table 2).* The presence of code smells has been correlated with higher change- and fault-proneness of code [82] and, thus, they could also be responsible for the triggering of refactoring actions. Also, static analysis tools are more and more used in the context of continuous integration to perform basic code quality checks at commit time. Thus, we decided to include the warnings raised by one of these state-of-the-art tools, i.e., PMD [13].

(3) *Process-related factors (Table 3).* These metrics are meant to provide a view on the development process, the developers involved in it, and historical information about the code components. We conjecture that these factors can play an important role in taking refactoring decisions, as also partially confirmed by previous work in the literature [104].

Table 2. Code Design Flaws (Product-related Factors)

Design Flaw	Description
DECOR Code Smells	
Blob Class	A large class that monopolizes most of the application logic [40]
Complex Class ✓	A class characterized by a high cyclomatic complexity [40]
Spaghetti Code ✓	A class declaring long methods without parameters [40]
CDSBP ✓	Class Data Should be Private: violation of information hiding principle [54]
Functional Decomposition ✓	Scarcely used object-oriented principles, such as inheritance and polymorphism; few methods and many private fields [40]
PMD code quality warnings	
Excessive Coupling ✓	A highly coupled class hindering reuse and maintainability [54]
Too Many Nested If Statements ✓	Makes the code harder to understand and increase error-proneness
Excessive Imports ✓	It might indicate too high coupling
Too High NPath Complexity	NPath is the number of acyclic execution paths throughout a method
Excessive Method Length ✓	It might indicate too many functionalities in a single method
Excessive Class Length	It might indicate too many responsibilities implemented in a class
Too Many Fields ✓	It can make the code hard to understand
Too Many Methods ✓	It might indicate too many responsibilities in a class
Cyclomatic Complexity	An excessive degree of decisional logic in a class
Excessive Parameter List ✓	It might indicate the need for a new object to wrap them
NCSS Type Count ✓	Similar to excessive class length, but it only considers actual statements
NCSS Method Count ✓	Similar to excessive method length, but it only considers actual statements
NCSS Constructor Count ✓	Equivalent of NCSS Method Count for constructors

Near each factor, we indicate whether (✓) it was retained. The factors retained in the model are also highlighted in bold.

Table 3. Process-related Factors

Metric	Description
Closeness to a previous release ✓	The number of commits until the previous minor/major release
Closeness to a next release ✓	The number of commits until the next minor/major release
Fault-Proneness ✓	Number of bugs fixed in the project history on a given class
Change-Proneness ✓	The average number of lines impacted in commits related to a class
Developer Overall Experience ✓	The number of past commits a developer performed
Developer Class Experience ✓	The number of past commits on a class performed by a developer

Near each factor, we indicate whether (✓) it was retained. The factors retained in the model are also highlighted in bold.

As detailed in Section 2.2.2, to avoid multicollinearity, we performed a variable selection. Near each metric, we indicate the cluster it belongs to and whether (✓) it was retained.

Source Code Quality Metrics. We consider, for each class C changed in each snapshot s_i, its quality *trend* as assessed by the 18 metrics in Table 1. These metrics capture different aspects of code quality, including size (e.g., ELOC), coupling (e.g., CBO), inheritance (e.g., DIT), complexity (WMC), encapsulation (e.g., NOPM), and readability (e.g., StrRead). The first 13 metrics in Table 1 (i.e., until NOSI included) have been computed by using the CK tool [4]. For the HsLCOM and C3, we used our implementation, while for the readability metrics we relied on the original implementations of the tools computing these metrics kindly made available by the original authors of the papers that introduced them [41, 90]. We start by measuring these 18 metrics on each class in each mined snapshot. Then, based on this information, we compute, for each snapshot, the slope of each

metric over a window of N preceding commits (we set $N = 10$ according to a previous work recommending just-in-time refactoring [84]). The slope of a line describes its steepness and in our case can highlight, for example, continuing degradation of some quality aspects (e.g., a high positive slope for the WMC metrics indicates a steep increase in complexity for a class over time). Thus, using slopes we capture the improvement or degradation of quality factors, where the latter may trigger a refactoring. Clearly, slopes were considered unavailable for the first 10 commits of a class.

Code Design Flaws and Quality Warnings. We consider code design flaws related to the lack of adoption of good Object-Oriented coding practices (i.e., Spaghetti Code, Excessive Coupling), to complex/large code components (i.e., Blob Class, Complex Class) as well as other design flaws and warnings (i.e., Excessive Imports, Too Many Methods) raised by a static analysis tool. We detect five types of code smells using an implementation of the DECOR smell detector based on the original rules defined by Moha et al. [77]. The choice of using DECOR is driven by the fact that (i) it is a state-of-the-art smell detector having high accuracy in detecting smells [77], and (ii) it applies simple detection rules that allow it to be very efficient. The latter was a strict requirement for our analysis, since we detected smells in all classes and for all studied systems' snapshots. In addition, we also consider 13 flaws from a widely used static analysis tool that does not require code compilation, i.e., PMD [13]. The set of detected design flaws and code quality warnings is described in Table 2.

Process-related factors. Besides the product-related factors previously described, we also study how process-related factors correlate with refactoring. In this case, we extract for each analyzed snapshot the factors summarized in Table 3.

Given a snapshot s_i, we compute its distance (in commits) from the previous and next release (first two rows in Table 3). This to verify the conjecture of Vassallo et al. [104] that refactoring does not occur immediately before/after a release. This information was retrieved using the GitHub API, through which it is possible to access all the tags related to a project. Then, we manually looked at the tags assigned to each project to isolate the ones referring to a new release.

We also consider the change- and fault-proneness of classes. The change-proneness is computed as the ratio between the total number of lines changed in the class C from the date of its addition to the project and the total number of commits in which C was changed, until each snapshot s_i.

The fault-proneness for C is computed as the number of bug-fixing commits it has been subject to in the past (i.e., before s_i). For each project, we first identified all bug-fixing commits by matching patterns [52]: "fix" or "solve" or "close" **and** "bug" or "defect" or "crash" or "fail" or "error." Then, for a given class C and for each snapshot s_i, we compute the number of bug-fixing commits preceding s_i and impacting C. Section 4 discusses the extent to which this simple heuristic for identifying bug fixes leads toward imprecisions.

Finally, we consider two metrics capturing the experience of the developers who worked on the system's classes. The first metric, named *Developer Overall Experience*, assesses the experience of each developer as the number of commits she performed in the past. For each snapshot s_i and for each of its classes C, we extract the list of developers who modified C in the past (i.e., before s_i). For each commit c_j (with $j < i$) in which C has been modified, we compute the experience of the developer authoring c_j (i.e., the number of commits she performed *before* c_j). This gives us a distribution of developers' experiences, for which we compute the minimum. Indeed, the minimum represents the lowest experience of a developer who worked on C, and we assume it might be correlated with future refactoring actions taken on C.

The *Developer Class Experience* computes a class-related experience: for each snapshot s_i and for each of its classes C, this form of experience is computed for a given developer as the number of commits impacting C she performed in the past. Thus, it is a more specific version of the overall

experience. We compute this metric for each s_i and C under study in the same way explained for the overall experience.

2.2.2 Metrics Aggregation and Preprocessing. Since in RQ_1 we are interested to build an *explanatory model* explaining which factors correlate with the presence of refactoring actions in a snapshot, we had to aggregate metrics for all classes involved in each snapshot. For the product metrics, we compute the maximum slope among all classes involved in the snapshot, except for the conceptual cohesion (*C3*) and readability (*StrRead* and *ComRead*) metrics, which go in the opposite directions than other metrics (higher values are better). In such cases, we consider the minimum. In both cases, the rationale is to identify the "worst case" in a snapshot, which could ideally trigger a refactoring. As for the DECOR smells, we count the number of classes exhibiting a smell in each snapshot, while for PMD we sum the number of warnings of each type among changed classes. Similarly to what done for product metrics, for process metrics, we compute the maximum (e.g., maximum number of bugs), except for the experience-related metrics, where we consider the minimum, again to consider the worst-case scenario. Finally, release-related metrics do not need to be aggregated, since they are already at commit granularity.

After that, to avoid multi-collinearity, we use the *R redun* function of the *Hmisc* package [57] for removing redundant variables. The *redun* function stepwise removes variable, starting from the most predicable one, until no variable can be predicted with an adjusted R^2 greater than a given threshold (0.8 in our study). Once again, we use the whole dataset to perform correlation analysis, because we intend to build an explanatory model and not a predictive model.

Since the value of our independent variables can depend on projects' characteristics, and to properly interpret the importance of each variable in the model, we normalize variable values, within each project, in the interval [0, 1]. This is done by subtracting the minimum and dividing by the difference between the maximum and minimum. Finally, to build a model easy to be interpreted, we invert (i.e., compute $1 - x$) the values of variables going toward a different direction than the others (i.e., those for which the higher the better).

2.2.3 Mixed-model Building. Once variables have been preprocessed, we address **RQ_1** by building mixed-effect generalized linear models. The model, built using the *glmer* function of the *lme4* [32] R package, is a logistic regression mixed-effect model where (i) the dependent variable is a dichotomous variable indicating whether at least a refactoring was performed in a given commit; (ii) the independent variables (fixed effects) are all the aforementioned ones, after having pruned out those highly correlating with others; and (iii) the random effect is the project in which the change occurred. The latter aims at controlling within-project effects, e.g., a project following a specific development process had better code quality assurance policies than others. To simplify, our model reports whether the status of the system (as assessed by the used independent variables) in the snapshot S_{i-1} triggered a refactoring in the subsequent commit C_i.

To answer **RQ_1**, we report the details of the model, among others the coefficient of each factor in the model, and the *p*-value indicating whether the factor is statistically significant or not (for a significance level of 95%). We also report the odds ratio (OR) that, for a logistic regression model, is given by e^{c_i}, where c_i is the coefficient of the *i*th factor. An OR > 1 indicates that a unity increase of a variable increases of OR times the chances of a refactoring to occur.

2.3 Qualitative Analysis of Refactoring Discussions in Pull Requests (RQ_2)

For the qualitative analysis, we identified PRs likely discussing refactorings using two criteria to be satisfied: (i) whether a commit is a part of PR or made during its review contains a refactoring identified by RMiner and (ii) whether the PR title or comments contain refactoring-related keywords. We used a list of refactoring keywords defined in a previous work [42] (available in our replication

package [27]) and augmented it with all names of refactorings identified by RMiner [99]. Note that, while this selection process can generate false positives (i.e., PRs unrelated to refactoring operations), these will be discarded during the manual analysis and, thus, do not represent a source of noise for our study.

Once the candidate set of 2,400 PRs has been identified, we created a randomly stratified sample of 551 PRs. The strata here were represented by the projects, i.e., PRs were sampled across projects proportionally based on the number of candidate PRs found in the previous step. The total number of PRs sampled allows us to ensure a significance interval (margin of error) of ±5% with a confidence level of 99%, and feature a total of 8,108 refactoring operations identified by RMiner. This estimation has been performed using a sample size (*SS*) calculation formula for an unknown population [88]:

$$SS = p \cdot (1 - p) \, \frac{Z_\alpha^2}{E^2}$$

and SS_{adj} for a known population *pop*:

$$SS_{adj} = \frac{SS}{1 + \frac{SS-1}{pop}},$$

where *p* is the estimated probability of the observation event to occur (we assume it to be 0.5 if we do not know it *a priori*), Z_α is the value of the *Z* distribution for a given confidence level, and *E* is the estimated margin of error (5%).

We then uploaded the sample of PRs on a tagging webapp we used to perform a manual coding of PRs. The webapp presented to the annotator the following information: (i) the PR title and hyperlink to the discussion, (ii) the refactoring-related keyword(s) matched in the PR text, and (iii) the list of refactorings detected by RMiner in commits linked to the PR, as well as the links to the GitHub diff pages of the commits themselves.

Through the coding app, each annotator could add one or more tagging items, containing the following information: (i) the type of refactoring action performed and discussed in the PR, or whether the change discussed was related to a combination of refactorings; (ii) whether the refactoring was the original intent of the PR, whether it happened as a consequence of the PR discussion or whether it happened accidentally because of another change; (iii) whether the refactoring was tangled with other changes, or if it was the only purpose of the PR; and (iv) finally, a tag indicating the motivation behind the refactoring, as it could be inferred from the inspection of the PR title/description, from its discussion, and from the commits related to it, looking at commit messages and, when needed, code diff. Note that each annotator could add more than one motivation for each PR (e.g., one for each refactoring operation, or even more than one for the same refactoring). To assign the tag describing the motivation, the annotator could choose an available tag in a drop-down menu (from those previously created by other annotators or by herself), or add a new one if no tag was fitting the specific case. If an annotator realized that the PR discussion was not related to refactoring, then the PR was tagged as "false positive."

Six of the seven authors took part in the annotation process. The webapp we developed took care of automatically assigning each PRs to at least two of the involved annotators. We collected a total of 1,223 tags each one reporting a motivation for a refactoring (or combination of refactorings) performed in a PR. After each PR was tagged by two annotators, three of the authors jointly worked on the available tags to perform a card sorting activity [94] aimed at merging duplicates (i.e., similar tags having the same meaning), and start grouping tags into categories. Then, they created a first taxonomy describing the different purposes of refactorings by only using the 699 tags for which there was no conflict (i.e., the same tag was used by the two annotators for motivating the refactoring observed in a PR). After a first draft of the taxonomy was produced, two different

Table 4. Generalized Mixed Effect Logistic Regression Model: Diagnostics, Residuals, and Random Effect

Diagnostics				
AIC	**BIC**	**logLik**	**deviance**	**df resid.**
21,071.5	21,362.4	−10,499.7	20,999.5	23,860
Scaled residuals				
Min	**1Q**	**Median**	**3Q**	**Max**
−2.0565	−0.5256	−0.3867	−0.1094	11.3848
Random effects				
Groups Name	**Variance**	**Std.Dev.**		
ProjectName (Intercept)	1.549	1.245		

authors refined it, by renaming some categories and moving sub-categories through the taxonomy. Once the final taxonomy was produced, three authors jointly discussed the conflicting cases in the categorization (524 of 1,223 tags) and assigned them to suitable taxonomy categories, creating new ones when needed, and ensuring a consistency of category naming.

To address **RQ₂**, we report and discuss the taxonomy of refactoring motivations inferred as previously explained. In particular, we discuss the various categories, highlighting the percentages of PRs belonging to the category, reporting some examples, and highlighting the implications resulting from our empirical findings.

3   RESULTS

In the following, we report and discuss the results addressing our RQs (Section 2).

3.1   Which product and process-related factors relate with an increase of refactoring operation chances?

Over the 213,102 snapshots analyzed, RMiner identified a total of 287,813 refactoring operations. More in details, our dataset contains 35,560 commits (≈17%) with at least one refactoring operation. If we exclude renaming operations (*Rename Method* and *Rename Class*), then RMiner found a total of 209,385 refactorings in 28,716 different snapshots (14%).

Table 4 and Table 5 reports the results of the logistic regression mixed-effect model. More specifically, Table 4 reports the model diagnostics (Akaike Information Criterion (AIC) [23], Bayesian Information Criterion (BIC), log likelihood, deviance, and degree of freedom residuals), the scaled residuals, and the random effect (project estimate). Concerning the model fitting (Table 4), we tried different models, namely logistic (i.e., the one reported, AIC = 21,071), linear (AIC = 22,565), and Poisson (AIC = 22,560). Also, although the analysis performed using the *redun* function already used a goodness-of-fit to iteratively remove variables, we experimented logistic models using structural metrics only (AIC = 89,751), conceptual metrics only (AIC = 89,475), code design flaws only (AIC = 179,528), and process metrics only (AIC = 23,583). Ultimately, the comprehensive logistic regression model we report is the one with the smallest AIC among those considered.

Table 5 reports the OR, estimate, standard error, *z*-value and *p*-value for the various factors we considered. We report in bold the coefficient for which there is a statistically significant correlation. Metrics that have been inverted (e.g., C3) are named with the prefix "Lack."

Looking at code quality metrics, we found that the lack of structural readability plays a significant role: lack of structural readability [41] increases the odds of refactoring (OR = 3.14). At the same time, *ComRead* readability metric and *LackC3* show a marginal significance *(p-value = 0.02 and p-value = 0.04)*, respectively. In particular, looking at the *ComRead* readability metric

Table 5. Generalized Mixed Effect Logistic Regression Model: Effect of Considered Factors

	Metric	OR	Estimate	Std. error	z-value	p-value
	(Intercept)	0.00	−7.52	0.52	−14.50	**<0.01**
Quality	**LackStructRead**	**3.14**	1.14	0.38	3.05	**<0.01**
	LackComRead	**2.68**	0.99	0.41	2.42	**0.02**
	LackC3	**1.87**	0.63	0.30	2.06	**0.04**
	CBO	1.02	0.02	0.03	0.66	0.51
	WMC	0.98	−0.02	0.01	−1.57	0.12
	DIT	1.17	0.16	0.12	1.39	0.17
	NOC	0.95	−0.06	0.28	−0.20	0.84
	RFC	0.98	−0.02	0.02	−1.05	0.29
	NOM	**0.88**	−0.12	0.05	−2.54	**0.01**
	NOPM	**1.22**	0.20	0.06	3.19	**<0.01**
	NOSM	0.91	−0.09	0.12	−0.76	0.45
	NOF	1.12	0.11	0.08	1.34	0.18
	NOSF	0.89	−0.11	0.13	−0.88	0.38
	NOSI	1.01	0.01	0.07	0.12	0.90
	LOC	1.00	0.00	0.00	0.91	0.36
	HsLCOM	**1.94**	0.66	0.29	2.28	**0.02**
Code Design Flaws	IsGodDecor	1.12	0.12	0.14	0.81	0.42
	IsCDSBPDecor	0.90	−0.11	0.15	−0.71	0.48
	IsComplexDecor	1.03	0.03	0.14	0.23	0.82
	IsFuncDecDecor	0.76	−0.28	0.25	−1.11	0.27
	IsSpaghCodeDecor	1.10	0.09	0.13	0.69	0.49
	AvoidDeeplyNestedIfStmts	0.92	−0.09	0.20	−0.43	0.67
	CouplingBtwObjects	0.73	−0.31	0.22	−1.42	0.16
	ExcessiveImport	1.04	0.04	0.20	0.18	0.86
	ExcessiveMethodLength	0.96	−0.04	0.23	−0.18	0.85
	ExcessiveParameterList	1.28	0.25	0.20	1.22	0.22
	TooManyFields	1.08	0.08	0.20	0.40	0.69
	TooManyMethods	0.66	−0.42	0.29	−1.42	0.16
Process	**LackGeneralExp**	**1.26**	0.23	0.10	2.30	**0.02**
	LackFileExp	**8.93**	2.19	0.13	16.45	**<0.01**
	FilesRelatedToIssueFix	**2.09**	0.74	0.07	10.12	**<0.01**
	AvgLinesImpactedInCommit	**1.93**	0.66	0.17	3.96	**<0.01**
	DistancePreviousRelease	1.13	0.12	0.08	1.59	0.11
	DistanceNextRelease	**1.43**	0.36	0.09	4.12	**<0.01**

combining structural and textual features [90] the OR is 2.68, while classes showing a decrease in their conceptual cohesion (*LackC3*) have 1.87 times higher odds of being refactored.

Among the structural metrics, we found that *NOM*, *NOPM*, and *HsLCOM* have a statistically significant effect (marginally significant for *HsLCOM*), although the OR for *NOM* and *NOPM* is close to one. Instead, the OR for *HsLCOM* is 1.92, indicating that, as expected, a lack of cohesion increases the odds of inducing a refactoring operation.

In conclusion, from our analysis it results that conceptual and readability metrics play a more important role in the model than structural metrics. This finding is aligned with previous work

aimed at applying conceptual metrics to suggest software refactoring [33] and modularization [38], and with findings of the seminal work about C3, indicating that such a metric is complementary to structural metrics [75].

None of the design flaws plays a statistically significant role. Although we expect that developers take care of removing smells, or try to "make static analysis tools happy," and although previous work has pointed the role of refactoring for improving code having a poor quality, e.g., overly complex code [39, 60, 91, 105], an evolutionary study on code smells indicate that smells mostly disappear when the source code is being rewritten, and only in less than 10% of the cases because of a refactoring action [101].

Interestingly, process-related metrics are highly representative if compared to product-related metrics: Five of the six considered process-related metrics are statistically significant. Previous bug fixes play a role: A unit increase of the *FileRelatedToIssueFix* factor results in 2.09 higher odds of applying a refactoring in the system. Not only classes subject to bug fixes are likely to be fault-prone in future [63, 79] but, since they are subject to (often quick-and-dirty) patches, they may necessitate refactoring actions. For related reasons, classes changing a lot (*AvgLinesImpactedInCommit*) also need to be refactored, although the OR is smaller (1.93).

Moving the attention to the metrics capturing the developers' experience, the *Developer Class Experience* (*LackFileExp*) has the highest OR. A unit increase of this factor related to the lack of specific experience (in terms of past commits) of developers that have recently modified a class, and therefore a decrease of experience increases the odds of refactoring by 8.93 times. In other words, changes applied by developers with little knowledge about a code component increase the need for restructuring it in the future. The general experience also plays statistically significant role, although the OR is relatively small (1.26).

Finally, looking at the proximity to a release (*DistanceNextRelease* and *DistancePreviousRelease*), the mectrics indicate that refactoring operations are applied to the system far from a release of the system. More specifically, increasing the number of commits to a subsequent release, there are 1.43 higher odds of applying a refactoring. However, results are not significant while looking at the number of commits from a previous release (i.e., *p-value* = 0.11). Our findings confirm previous literature [61, 104], since developers are aware that some kind of refactorings may result in the introduction of new faults [34] and in any case, refactoring represents a costly and risky operation [61]. For this reason, it is not very common to apply refactoring close to a new release of the software product. Furthermore, once released a new version of the software, developers likely tend to focus on bug-fixing activities instead of applying refactoring operations. In summary, based on our observations, refactorings are less likely to occur either immediately before major releases (developers focus on new features and, for what possible, on reliability of what they release), and immediately after (developers work on bug fixes). Instead, refactorings are more likely to happen in-between. Once again, note that this conclusion is based on purely observational data, and distance from the next release is unlikely to be used for prediction purposes (developers do not know when the next release is, unless a project or organization adopts very rigid release timelines).

We conclude RQ$_1$ stating that, on the one hand, by observing product metrics, only code readability plays a significant role. On the other hand, process-related metrics play a significant role. These are metrics related to previous changes and bug fixes, and to the experience of recent change authors.

3.2 What are the reasons for performing a refactoring operation?

Before digging into the results Table 6 reports statistics about the refactoring operations we found in the analyzed PRs. Note that (i) several refactorings can be applied in one PR, therefore the number of refactorings is higher than that of PRs and (ii) we only list refactoring operations we

Table 6. Statistics of Refactoring Operations Labeled in the 551 Analyzed PRs

Refactoring operation	Freq.	When?			Tangled	
		Orig. intent	Collateral	After discuss.	Yes	No
Combination of refact.	578	65%	3%	32%	86%	14%
Extract operation	112	62%	3%	35%	62%	38%
Rename method	69	47%	4%	49%	59%	41%
Rename class	53	39%	6%	55%	56%	44%
Move class	39	64%	8%	28%	49%	51%
Extract variable	29	52%	35%	14%	72%	28%
Rename variable	29	35%	14%	52%	83%	17%
Extract interface	27	59%	11%	30%	59%	41%
Rename attribute	22	35%	8%	57%	44%	56%
Extract and move	22	68%	0%	32%	55%	45%
Rename parameter	19	69%	5%	26%	37%	63%
Move attribute	15	53%	14%	33%	73%	27%
Move operation	13	69%	0%	31%	39%	61%
Extract superclass	10	70%	10%	20%	80%	20%
Overall	**1117**	**60%**	**5%**	**35%**	**73%**	**27%**

The "Freq." column reports the number of times that the annotators defined a tag explaining the rationale behind each specific type of refactoring operation. The total number of 1,117 tags is the result of the 1,223 tags we defined, excluding the 94 "unclear" (i.e., cases in which the annotators did not manage to identify the rationale for the refactoring) and 12 "false positives" (i.e., PRs that were unrelated to refactoring).

observed at least 10 times. However, the overall number of refactorings (1,117) also includes the instances related to refactoring types we do not show in Table 6 (since having fewer than nine occurrences). In most cases, developers do not discuss a specific refactoring operation. Instead, they rather provide a rationale for a combination of refactorings (~52% of the cases). Then, *extract operation* (~10%) and renaming refactorings in general (~17%, in total) are the ones more discussed by developers. Surprisingly, we found that only a small percentage (5%) of refactorings were done collaterally, i.e., without mentioning them at all. Instead, many of them were done as the original intent of the PR (~60%) or after discussing with other developers (~35%).

Some refactoring operations, such as *extract variable* and *rename variable*, were performed collaterally more often, given their "local" nature: A variable name only matters in the methods in which it is declared, while a class name can possibly impact the whole system. It is worth noting that developers perform most of the renaming operations after they receive feedback from their peers. This shows that names are often discussed in PR reviewing activities. Finally, in line with Murphy-Hill et al. [80], we found that about a fourth of the refactorings are tangled with other changes.

Figure 2 depicts the taxonomy of refactoring motivations we have identified. It comprises six root categories: (i) *Improve Code Design* groups refactoring operations targeting an improvement of the system design, e.g., by fostering the reusability of code; (ii) *Improve Understandability & Readability* includes refactorings aimed at reducing the effort to read and understand code, e.g., by renaming identifiers; (iii) *Improve Quality of Test Code* groups all refactorings performed to improve the quality of the test code or ease the testing process; (iv) *Prevent Bugs* identifies refactorings performed to prevent the future introduction of bugs; (v) *Preparing Code for Changes* includes refactorings performed in preparation of other changes, e.g., refactoring the code before implementing a new feature; and, (vi) finally, *Other Motivations* groups those motivations that cannot be classified into one of the previous categories.

Fig. 2. Motivations behind refactoring operations.
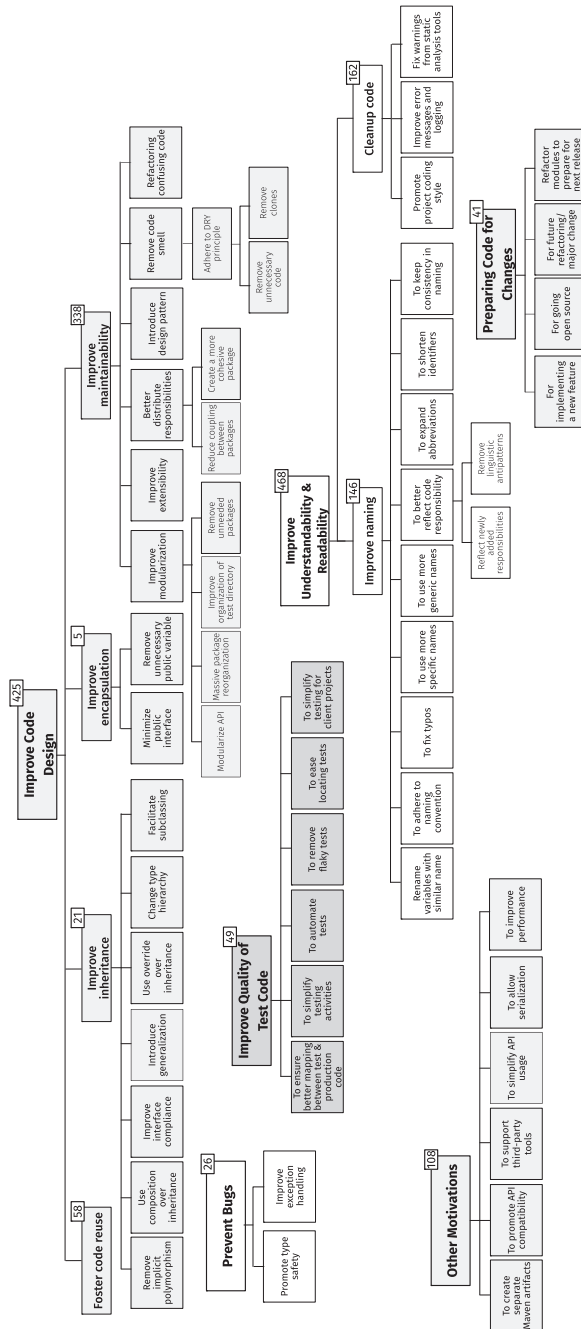
It is important to point out that some of the categories of the taxonomy are not mutually exclusive. For example, a refactoring aimed at improving code readability, is also likely to improve maintainability. However, readability can be improved for multiple purposes (e.g., simplify testing), and, for this reason, we separated these categories. We acknowledge that other choices in terms of categories and assignment of instances to these categories are possible. Also, the hierarchical organization of the categories only indicates that child categories are specialization of their parent categories, while it does not imply that two categories at the same hierarchy level represent motivations at the same level of abstraction (e.g., *Improve Inheritance* and *Foster Code Reuse* appear at the same level, but the former is a more concrete motivation as compared to the latter).

Figure 2 also reports for each category of "motivations" the number of PRs in which we found related refactoring operations. For readability purposes, we only report these numbers for the main categories. Note that the number for a parent category does not correspond to the sum of the children, because some PRs were only assigned to the parent category, as the motivation was not specific enough. Also, the sum of refactoring instances in all root nodes does not correspond to the total number of 551 manually analyzed PRs, because some PRs comprise refactorings falling into multiple categories, and we labeled some refactorings as *Unclear* (94) and discarded 12 PRs as *False Positive*.

We compared our taxonomy with the list of 44 motivations derived by Silva et al. [91] for 12 frequently applied refactoring operations (see Tables 3 and 4 in Reference [91]). In particular, two of the authors tried to map Silva et al.'s motivations into our taxonomy, to see whether they were covered or not. Note that the mapping is not one-to-one, since one motivation identified by Silva et al. [91] may be mapped to more than one category in our taxonomy, as well as one of our categories can group more than one of their motivations. This is expected, since their motivations and the categories in our taxonomy have been derived by using two different methodologies. Indeed, while we have categorized the possible motivations behind the application of refactoring operations by looking at the discussions in PRs, Silva et al. [91] have asked the reasons behind specific instances of refactoring operations to the original developer who has applied it.

Only 3 of the 44 motivations from Silva et al. [91] cannot be mapped in our taxonomy. The main reason is that for these three instances (i.e., *Enable recursion*, *Convert to top-level container*, and *Convert to inner class*) it was unclear to us the actual motivation behind the refactoring. For example, enabling recursion could be done to improve performance as well as to improve code readability. However, this high overlap between the two sets of motivations (i) validates and generalizes the work done by Silva et al. and (ii) supports the comprehensiveness of our taxonomy. As reported in Table 7, our taxonomy features 16 inner categories that are not covered in Ref. [91] (e.g., *To Ensure Better Mapping Between Test And Production Code*), 3 inner categories that are only partially covered (e.g., *Preparing Code for Changes*), and 6 inner categories (e.g., *Forster Code Reuse*) that are completely covered. We provide in our replication package [27] a spreadsheet reporting the mapping between the two taxonomies.

In the following, we discuss each root category, reporting interesting examples and outlining implications for researchers and practitioners (indicated with the ♀ icon), as well as highlighting the differences with the taxonomy provided by Silva et al. [91]. The complete list of manually analyzed PRs together with their refactorings/assigned tags is publicly available [27].

Improve Code Design (425 instances). Unsurprisingly, a large proportion of the analyzed refactorings are aimed at improving code design from several perspectives [55]. In 58 of these, the refactorings are aimed at making source code easier to be reused (see *Foster code reuse* in Figure 2). In 42% of these cases, this was accomplished through a combination of several refactoring operations, while in the remaining 58% specific refactorings were applied in isolation. When this happened, almost always (91%) an operation aimed at extracting a code component from an

Table 7. Comparison with Refactoring Motivations Found by Silva et al. [91]: ↑↑ Highlights
a Perfect Match (the Motivation Also Emerges from Their Study); ↑ Highlights a Partial
Match (We Found Additional, Specific Motivations); ↓↓ Stands for a Mismatch
(the Motivation Was Not Found in Their Study)

Root Category	Inner Category	Match
Improve Code Design	Foster code reuse	↑↑
	Improve inheritance	↑
	Improve encapsulation	↓↓
	Improve maintainability	↑↑
Improve Understandability & Readability	Improve naming	↑
	Cleanup code	↓↓
Improve Quality of Test Code	To ensure better mapping between test and production code	↓↓
	To simplify testing activities	↑↑
	To automate tests	↓↓
	To remove flaky tests	↓↓
	To ease locating tests	↓↓
	To simplify testing for client projects	↓↓
Preparing Code for Changes	For implementing a new feature	↑↑
	For going open source	↓↓
	For future refactoring/major change	↓↓
	Refactor modules to prepare for next release	↓↓
Prevent Bugs	Promote type safety	↓↓
	Improve exception handling	↓↓
Other Motivations	To create separate Maven artifacts	↓↓
	To promote API compatibility	↑↑
	To support third-party tools	↓↓
	To simplify API usage	↓↓
	To allow serialization	↓↓
	To improve performance	↑↑

existing one was applied. In particular, *extract method* operations were performed in 70% of cases, to extract a small piece of functionality from an existing method thus avoiding code duplications and fostering the reuse of the extracted code. For example, during the code review of the PR #626 in the nakadi project [20], the reviewer observed that two of the implemented methods were "*almost the same except the very last line*" and suggested to "*extract a helper method*" in such a way to reduce code duplication and also allow other methods, in future, to reuse the same functionality. This was accomplished through an extract method refactoring. In other cases, the refactoring was more substantial and directly justified by the need for reusing specific pieces of functionality, as discussed in the PR #488 of the dropwizard project [6]. Here the contributor explains, when submitting the PR: "*I was looking at starting/stopping a Dropwizard app in Cucumber tests and DropwizardAppRule has all the functionality I need but obviously it doesn't expose startIfRequired and stop methods. I'd happy to extract a DropWizardAppTestSupport class from DropwizardAppRule.*" After approval, this triggered an *extract class* refactoring. This last example is interesting for several reasons. 💡 First, extract class is a non-trivial refactoring possibly having substantial ripple effects in the system, with the obvious possibility of introducing bugs. For example, the discussed commit impacted a total of 499 lines of code, thus showing that code reuse is a strong motivation for triggering refactoring

operations. Second, while many approaches to identify extract class refactoring opportunities have been proposed (see, e.g., References [35, 53]), they focus on the identification of complex classes implementing several responsibilities (i.e., *God* or *Blob* classes [40]) that could be split into several classes. The class subject of the *extract class* refactoring (i.e., `DropwizardAppRule`) is a fairly simple class composed by 156 effective LOC (excluding comments and blank lines) that is unlikely to be reported by refactoring recommenders as a candidate for *extract class* refactoring. To cope with these cases, these recommenders could be combined with clone detectors [89] to factor out a class to be used by multiple other ones. Note that these "special" cases should complement the more standard refactoring recommendations done for complex and low-cohesive classes. Indeed, as shown in our RQ$_1$, classes characterized by a low cohesion as assessed by the C3 and HsLCOM metrics are more likely to be subject to refactoring operations.

Many (338) of the refactorings performed in the code design taxonomy aimed at *Improve Maintainability* (see Figure 2). In this category, refactorings aimed at improving the modularization were often implemented through simple move class refactorings, while we rarely observed massive package reorganizations (7 cases). 💡 This is in line with recommendations from previous literature, suggesting that approaches performing big-bang remodularization through clustering algorithms have limited applicability, and techniques suggesting fine-grained and incremental adjustments to software modularization should be preferred [56, 81]. Also, it was interesting why developers decided to perform remodularization. For example, in some cases move class refactorings are performed to group, in specific "API-related" packages, utility classes potentially useful in different parts of the system and/or to third-party components, e.g., PR #324 from DSpace "*I suggest to move this class in dspace-api as it will be useful to port this feature to JSP UI as well*" [8]. While these changes might look suboptimal from the cohesion-coupling point of view (i.e., they could generate a low-cohesive package) they are justified by a clear rationale. 💡 As also observed for the approaches automating *extract class* refactoring, tools recommending modularization solutions (see, e.g., References [28, 65, 73, 86]) just strive to maximize the cohesion-coupling tradeoff. Given the availability of historical data, they could also learn from previous changes what a meaningful modularization is from the developers' perspective. While learning code changes is already an active research field [102], no previous work has attempted to design refactoring recommenders learning from developers' activities what a meaningful refactoring is in a given context.

Removal of code clones is one of the two motivations behind the refactorings in the *Adhere to DRY principle* (Don't Repeat Yourself) subcategory (child of *Remove Code Smell*), together with the removal of unnecessary code. Note that this category is strictly related to the *Foster code reuse* one. Indeed, some of the analyzed PRs fall into both these categories, because factoring out duplicates also creates a more generic code element (e.g., a class or method) that can be further reused. For example, PR #366 in the `fineract` project [2] can be seen as an example of improving reusability by adhering to the DRY principle, since it features an *extract method* refactoring suggested by the reviewer and avoiding code duplication while allowing the reuse of a piece of functionality now embedded in the extracted method. 💡 This confirms once more the relevance for practitioners of clone detectors [89] as well as of refactoring tools aimed at removing clones [100] and encourages their use in the Continuous Integration (CI) pipeline, as advocated by Duvall et al. [50].

Concerning the removal of unnecessary code, besides cases simply related to removing unused `imports`, we found refactorings performed to remove redundant code (e.g., PR #1481 from `testng` [3]. 💡 While some work has investigated the automatic identification of redundant code in software systems [59, 70], the provided support is still very limited to specific redundancy cases (e.g., those related to API usages [59]) or programming languages (e.g., LISP [70]). The 55 cases related to refactorings motivated by the removal of unnecessary code suggest room for more research in this field.

Concerning the operations targeting a better distribution of the responsibilities across code components, one very interesting example comes from the DSpace project (PR #1083 [7]). This PR implements a massive refactoring aimed at ensuring a better "separation of concerns/responsibilities" for an API module, and has been subject to votes by the community, because the refactored API was not backward compatible. Despite this issue, the merging has been approved thanks to the numerous positive advantages brought by the refactored API: "*makes it much easier to achieve future goals on our Roadmap, especially, moving us toward potentially better support of third-party modules,*" "*it cleans up one of the messiest areas of our existing API [...]*". 💡 This case shows the non-trivial trade-offs that developers should consider in case of massive refactoring: for example, smartphones have limited battery life and they require software optimized to reduce the energy consumption. State-of-the-art refactoring recommenders [35, 98] ignore the heterogeneity of modern software, and the different priorities that non-functional requirements, possibly more important (e.g., maintainability, performance, backward API compatibility) may have in different contexts. Future work should consider integrating into these recommender systems the possibility to define a priority list of non-functional properties that developers are or are not willing to sacrifice when applying refactoring. This would allow generating more meaningful and sensible refactoring recommendations.

The two sub-categories described above (*Foster code reuse* and *Improve maintainability*), i.e., the ones highly represented in our study, have a complete matching with the motivations identified by Silva et al. [91]. This confirms their findings, and stresses once more the importance from the developers' perspective of improving both the reusability and maintainability of code, especially when discussing whether to accept or not a PR.

Other less represented subcategories in the *Improve code design* taxonomy include refactorings aimed at improving the usage of inheritance (21 instances) and the ones working on the encapsulation (5). In these cases, considering the low number of instances belonging to each category, it is quite obvious that while comparing with the motivations by Silva et al. we found that these reasons did not emerge from their study. The only exception is the one related to inheritance that is only partially covered, as shown in Table 7.

Improve Understandability and Readability (468 instances). The majority of refactorings we found in the manually analyzed PRs aim at improving understandability and readability of source code. This supports the findings of RQ$_1$, which indicate a significant correlation (and high OR) of readability metrics with refactoring operations. In this category, 146 refactorings were done to improve naming. The observed renamings had a variety of motivations (see Figure 2), ranging from fixing typos to keeping naming consistency throughout the project. Naming decisions were often carefully discussed, showing their importance for developers. An interesting example of discussion about naming is the PR #150 of the optaplanner project [11]. The original intent of the PR was to add a new feature, but the author explicitly asked for feedback about the naming of a new interface he extracted: "*We should discuss the naming and the usage of the SolverProblem-BenchmarkResult interface.*" Such a name was changed after the discussion: "*renamed to BenchmarkResult as agreed in a meeting.*" In the same PR the developers also discussed several other names in the contributed code. For example, the PR author introduced a boolean field named hasNonDefaultSubSingleCount; another developer asked why such a value was introduced, since the name was not clear enough. The discussion triggered not only a renaming operation, but also a type change (from boolean to integer) to represent additional information that could be useful in future: "maximumSubSingleCount*is the best name here, as it gives us more potential information for the future at no cost*".

We also found cases of renaming aimed at better reflecting the code responsibility (38). A representative example is in PR #251 of the kafka-connect-elasticsearch project [5], in which one of the reviewers suggested to rename a test method to something very specific and clearly

depicting the responsibility of the test case: "*you could change the name of the test method to something like testCreateAndWriteToIndexForTopicWithUppercaseCharacters. I like test names that read like the condition they are testing*". Other cases aimed at removing linguistic antipatterns defined in the literature by Arnaoudova et al. [31] and known to have negative effects on the understandability of code [51]. ♀ This is only one of the studies linking the poor quality of identifiers to difficulties experienced by developers in code comprehension [48, 67, 68, 69, 97]. Our findings show that developers care about the quality of identifiers and carefully discuss their choice.

♀ Most of the rename refactoring recommendation approaches aim at fostering the usage of consistent naming [24, 71], while only a single attempt has been done, to the best of our knowledge, to recommend rename method refactorings with the goal of better reflecting the responsibilities implemented by the code [25]. Our results show that more effort in this direction is needed, since this is the scenario in which developers more frequently perform rename refactorings. Also, ♀ the high number of rename refactorings implemented as a consequence of code review indicates the possibility to mine this data to evaluate automated rename refactoring techniques [24, 71]: the originally submitted identifier represents an opportunity for rename refactoring while the one adopted after the code review process can be used as reference of a good refactoring. This would avoid the evaluation of the rename refactoring techniques in artificial scenarios.

Looking at Table 7, and considering that developers can modify the names of packages, classes, variables or methods for different reasons, we can state that our *Improve Naming* category is only partially covered by the motivations reported in the previous study by Silva et al. [91]. For instance, while both studies identify the need for adhering to naming conventions, for keeping consistency in naming, or for better representing code responsibilities, in our taxonomy we also found cases where the renaming occurs to fix typos, shorten identifiers and expand abbreviations. ♀ The latter challenge (i.e., expansion of abbreviations in code identifiers) has been vastly investigated in the software engineering research literature (see, for example, the works by Lawrie et al. [66]), with approaches proposed and empirically evaluated. However, to the best of our knowledge, there are no ready-to-use tools that, for example, can be integrated in a CI pipeline and can recommend to developers identifiers to expand at commit time. The implementation of such a tool is a clear next step to perform in this research field.

We also found several refactorings implemented to make the source code less confusing. Such changes involved improvements to both names and structural aspects. For example, we found an interesting example in PR #599 of the `htsjdk` project [15]. Note that the refactoring performed in this PR is an *extract interface* rather than a rename, that resulted in the interface `CRAMReferenceSource` implemented by the class `ReferenceSource`. The main goal of the refactoring was to improve code reusability: For this reason, we include such a case in our taxonomy under the *Foster code reuse* category. However, it is interesting to discuss this refactoring in the context of renaming: during the code review process, one of the reviewers argued that the chosen names were confusing, because he expected an inheritance relationship in the opposite direction (i.e., `CRAMReferenceSource` implements `ReferenceSource`) by reading the names alone. As a consequence, `ReferenceSource` was renamed to `CRAMReferenceSourceImpl`, making the relationship between the two classes more evident. ♀ This naming issue could be characterized as a sort of linguistic antipattern, and shows that the original catalog of these antipatterns defined by Arnaoudova et al. [31] could be expanded by analyzing recommendations provided by reviewers in a code review process.

Finally, we found many cases in which the developers made more generic clean-ups in the code (162 instances), to improve the coding style and, in some cases, the quality of error messages and logging. Some of these changes were performed as a result of tools' recommendations. For example, the PR #346 of the `spring-amqp` project [18] fixes warnings raised by SonarQube. ♀ This suggests

that developers are willing to fix issues identified by automatic tools. However, in our sample of PRs, SonarQube was the only tool mentioned in many discussions. Note that a specific analysis of the extent to which static analysis tool warnings are removed was not in scope of our work (but rather addressed in related literature [47, 62, 92]); this is the reason why, in RQ_1, we only considered two tools—DECOR and PMD—that could raise warnings that triggered refactoring operations.

More than 60% of the clean-up operations were done as part of the original intent of the PRs. Differently from other categories, we found very little discussion among developers regarding clean-ups. This is likely due to (i) the limited impact that these clean-ups generally have; and (ii) a general agreement on the need for improving code quality.

Prevent Bugs (26 instances). These refactorings are motivated by the will to prevent bugs, for example through a better exception handling or by promoting type safety. Note that these are changes that preserve the program's behavior. For this reason, we contemplated them in our taxonomy, even if they do not belong to the canonical cases of refactorings, such as those defined by Fowler [55]). This is why those categories are completely uncovered in the 44 motivations provided by Silva et al. [91]. Indeed, they studied the reasons behind specific refactoring operations that are detected by RMiner and inline with those defined by Fowler [55].

Some PRs are explicitly motivated by the will of improving the exception handling mechanism. This is the case for PR#933 of the nakadi project [21], in which the developer implements several different refactorings (e.g., *rename class*, *move class*) to improve the overall handling of the exceptions in the project. Other PRs, instead, simplify the handling of exceptional conditions. For example, in PR#1067 from the htsjdk project [14], the developer fixes a possible issue caused by the invocation of the method mFile.getSource() within several exception messages. Indeed, in specific cases the mFile object could be null, leading to the throwing of a NullPointerException. For this reason, the developer implemented an *extract method* refactoring, creating the method getSource(), which returns the value of mFile.getSource() when mFile is not null, and a constant string otherwise. This allowed to easily prevent NullPointerException by replacing the many usages of mFile.getSource() with an invocation to the newly created getSource() method. 💡 This is an interesting application of *extract method* refactoring, since it aims at refactoring a very small clone, i.e., a method invocation reused, in the same way, in different parts of the code. Extract method is widely applied in the refactoring of clones [64]. However, the focus is usually on more complex clones sharing several statements, rather than on the identification of refactoring opportunities that, as in the discussed case, involve few code tokens but can have a positive impact on the reliability and maintainability of the system.

Another very interesting example is the PR#238 from the minio-java project [12]. Here the developer replaced general, unchecked exceptions such as NullPointerException, with more specific and checked ones. With "unchecked," we refer to those exceptions that in Java can be thrown without declaring them in the method signature. For example, in the specific case of PR#238, the method getClient of the class Client was throwing a NullPointerException in specific situations: note that this was an intended behavior of the method, i.e., there was an explicit throw new NullPointerException() in the code. However, in the method signature the only visible exception was MalformedURLException. Through the implemented changes, including *class rename* refactoring, the more general exceptions have been specialized (e.g., to ClientException in the case of the getClient method), forcing the exposure in the signature of the thrown exception. This has the double effect of (i) giving developers compilation errors if they do not catch the thrown exception, thus preventing bugs and (ii) using more expressive exception names. 💡 Note that the usage of unchecked exceptions in Java code should not be considered as a "bad smell," since, in general, unchecked exceptions should be used to reveal bugs, while checked exceptions to throw errors that the program should handle [45]. However, the misuse of unchecked exceptions where

checked ones are needed can lead to higher chances of introducing bugs (as in the case of PR#238). The automatic identification of these situations is, to the best of our knowledge, a problem still not faced in the research literature.

Preparing Code for Changes (41 instances). This category includes refactorings facilitating the implementation of new features or of other planned activities. Refactorings preparing for future changes are usually implemented in dedicated PRs including major refactorings.

Looking at the comparison highlighted in Table 7, it is possible to state that our taxonomy provides more insights compared to the list of motivations in the previous study. Indeed, the category *Preparing Code For Changes* contains some motivations already highlighted in Reference [91] while others missed such as the need for refactoring operations aimed at moving to open-source. An interesting case in this category is represented by PR #317 of the sagan project [17]. As mentioned in the title, the goal of the PR is to *"refactor in preparation for open source."* Note that this is kind of an exception in our taxonomy, and we decided to put it into the *Preparing Code for Changes* category just because open sourcing a project is, in some way, a decision taken to foster the future evolution of the project. Code refactoring is one of the action items in a checklist defined in issue #179 for preparing the project to be open-sourced [16]; other action items included, for example, the introduction of installation and configuration instructions. 💡 This suggests that an appropriate code cleanup, including refactoring, should be part of packaging checklists when putting a project in the open-source.

Another example of refactoring performed to accommodate other changes is in PR#136 of the zhcet-web project [22], where the developer extracted the class CryptoUtils from SecurityUtils to accommodate the implementation of new functionalities (e.g., the decrypt method) in a suitable class (i.e., CryptoUtils).

💡 Tools supporting preemptive refactoring are lacking in the literature. Indeed, the only effort in this direction is the work by Pantiuchina et al. [84] in which, however, the focus is on identifying classes that will be affected by code smells in the future, thus recommending them for a preemptive refactoring action. 💡 Our manual analysis indicates that a novel family of recommender systems able to suggest developers how to refactor the code to "accommodate" the implementation of a given change request could be valuable.

Improve Quality of Test Code (49 instances). As the production code might be in need for refactoring, this also holds for test code [103]. The quality of the test code is also assessed in the context of PR discussion and code reviews, as observed by Spadini et al. [93]. We found 49 test code refactoring cases, 37% of which performed with a specific type of refactoring operation, and 63% with multiple operations. The observed refactorings include changes similar to those performed on production code, e.g., better distribution of responsibilities to have a better mapping between test and production code, see, e.g., PR #1071 in the error-prone project [10] in which the author comments "*[…] separate the tests into logical classes.*"

Other cases we found concern the removal of flaky tests that introduce non-determinism in test outcome [76]. In the microprofile-fault-tolerance project, the PR #363 [9] aims at removing flaky test: "*The tests testCircuitInitialSuccessDefaultSuccessThreshold and testCircuitLateSuccessDefaultSuccessThreshold were moved to an independent test to avoid dependencies between tests that use the same bean […] that can generate possible failures when the circuit breaker leaves open […].*"

More interesting and specific for test code are the refactorings performed to improve testability. In PR #73 of the WPS project [1] a developer performed an *extract class* refactoring in production code motivated by the will of simplifying integration testing: "*The functionality to create a GTVectorDatabinding out of shapefiles was removed from the GenericFileData class and moved to a new GenericFileDataWithGT class. Due to this change, the processes used for the integration tests do not depend on GeoTools anymore […] also the tests now use only local resources.*" 💡 This

example shows how refactoring performed on production code can have an impact on many software quality aspects (in this case, testability). Cases like this one suggest to always ponder the positive and negative impacts of refactoring beyond maintainability, e.g., some refactoring actions can aid testability, while others might improve maintainability at the cost of testability. ♀ Clearly, considering this aspect in the context of a refactoring recommender is far from trivial, given the need for automatically assess the testability of a given component. Besides, the presence of refactorings specifically aimed at improving testability shows room for approaches aimed at recommending such kind of operations.

While the motivations of refactorings for simplifying testing activities were already presented in the work by Silva et al. [91], our taxonomy provides other new categories such as those aimed at removing flakiness, at automating testing activities, or at improving the overall testability of the project under development.

Other Motivations (108 instances). In this category, we put all motivations that did not find their place among other root categories.

Thirty-seven PRs were performed to improve software performance. This is not surprising and in line with Fowler [55], who observed that the internal program structure is closely related to its performance due to better optimization opportunities. Moreover, the latter also emerges from the motivation provided in the previous study by Silva et al. [91]. Also, our quantitative results of RQ_1 indicate that refactorings have a high chance to occur on classes frequently subject to bug fixes, which may have affected performance, especially in the case of quick patches. A concrete example is the PR #1577 of `AmazeFileManage` [19] Android application. In a linked issue, a user reports that when she is "*copying a large file using SFTP, the process can take more than 1 minute, so the phone goes in stand by mode.*" PR #1577 improves the I/O performance of this feature through refactoring.

♀ This example confirms the importance of specific non-functional attributes (in this case, performance) for different types of software (in this case, a mobile app). Also, once again, it points to the need for developing refactoring techniques able to consider this heterogeneity of non-functional requirements rather than mainly focusing on maintainability as done in state-of-the-art refactoring tools [35, 98]. Indeed, to the best of our knowledge, only a few authors have developed refactoring techniques having the improvement of performance as the main objective [30, 49, 106]; however, these approaches either target very specific performance issues [49, 106] or are designed to work on models rather than on source code [30].

4 THREATS TO VALIDITY

Construct validity. A source of inaccuracy is represented by the automated refactoring detection. However, RMiner has been reported to exhibit a very high precision (98%) and recall (87%) [99]. This threat is mitigated at least in RQ_2, where refactorings have been manually reviewed. To identify bug fixes, we used an approach matching regular expressions onto commit messages [52], as also done in previous work [87]. To limit threats due to this heuristic [29], two authors independently analyzed, for each project we considered, a random sample of commits classified as a bug fix to mark true and false positives. After discussing disagreements, only 8% of the analyzed commits resulted to be false positive bug fixes (mostly related to CheckStyle fixes).

In RQ_1, we only analyzed the correlation between the presence of any refactoring with various metrics. While it may be interesting correlating specific types of refactorings with metrics, our qualitative analysis showed that refactoring goals are often achieved through a combination of refactorings. To build the explanatory model of RQ_1, we have selected a broad set of metrics capturing different aspects of software product and process. It is important to note that the aim was to correlate such metrics with the presence of at least one refactoring action of any kind. Building

models for specific refactoring types is out of scope of this article and could, possibly, require to identify further specific indicators.

In RQ_2 we identified refactoring-related PRs as those having (i) one of their commits containing a refactoring identified by RMiner and (ii) a refactoring-related keyword in their title. Such selection criteria can result in false negatives (i.e., missing some refactoring-related PRs) and, in turn, this may have resulted in missing categories in our taxonomy. Indeed, it is possible that our taxonomy is only representative of the motivations behind PRs that can be captured through the adopted selection criteria.

As context for our study we targeted non-personal/toy projects having a substantial change history to study and being active. For this reason, we defined a number of selection criteria (i.e., at least 5 contributors, 1 fork, 500 commits, 100 PRs, and one recent commit) that, however, may fail in capturing the type of systems we were interested in.

Internal validity. In the quantitative analysis (RQ_1), although we tried to capture factors from different dimensions (i.e., different kinds product and process metrics), there could be many other factors that could have influenced the need for refactoring. We mitigated this threat through (i) the use of a mixed-model considering project as random effect and (ii) the qualitative analysis of RQ_2. It is important to note that the aim of RQ_1 is to mainly identify correlations between metrics and refactoring activities, and not about claiming any causation. Only the qualitative analysis of RQ_2, taking into account developers' discussion, can highlight the rationale for refactoring actions.

Conclusion validity. In RQ_1 we performed a careful preprocessing of data and variable selection to avoid multi-collinearity, and normalized metrics to allow properly interpreting the ORs.

External validity. Our analysis is limited to a sample of 150 Java open source projects hosted on GitHub, and the qualitative analysis to 551 PRs. We do not claim the generalizability of our findings to other programming languages or to industrial systems. For this reason, a further investigation on a more diverse set of projects, developed with different programming languages and belonging to both open and closed-source is highly desirable. Also, it is worth mentioning that in our manual analysis we only considered PRs for which RMiner identified at least one refactoring operation. This means that we did not consider PRs that, for example, targeted a complete remodularization of the system that involved refactoring operations not captured by RMiner.

5 RELATED WORK

Different studies have empirically analyzed refactoring operations from different perspectives, including how developers perform refactoring [80]; the relationship between refactoring and other software-related activities (e.g., merge conflicts [72]); the impact of refactoring operations on the likelihood of introducing bugs [34]; the impact of refactoring on specific quality indicators (e.g., quality metrics) [26, 44, 95, 96], or on developers' productivity [78]. While these studies mine and analyze refactoring for different purposes, they address different research questions as compared to the ones subject of our work. For this reason, we focus the discussion on studies analyzing why developers perform refactoring.

Wang et al. [105] interviewed 10 developers from four software companies to reveal the major factors motivating refactoring operations. Results highlight external motivators e.g., *Recognitions from Others*, and intrinsic motivators, i.e., when refactoring is initiated without any obvious external reward (e.g., *Self Esteem*) behind refactorings.

Kim et al. [60] present a field study surveying and interviewing 328 Microsoft engineers to investigate when and how engineers refactor code. They identify the low readability of source code as the most important symptom that pushes developers to perform refactoring (mentioned by 21% of developers). Our quantitative analysis confirms the central role of low code readability

in triggering refactoring. Other frequently mentioned symptoms were code duplication (11%), and fostering code reuse (10%).

Silva et al. [91] observe that the previously discussed works [60, 105] report findings from surveys asking developers about their general refactoring habits, without focusing on real refactorings they performed. Thus, they use RMiner [99] to monitor refactorings performed in open source repositories and contacted the developers authoring the refactorings asking to motivate the performed changes. Then, they grouped the responses and defined a catalog of 44 motivations for 12 refactoring operations. For example, they find that three main motivations exist for *Rename Package* refactoring: Improve the package name, enforce naming consistency, and move package to appropriate container [91].

While we share with previous work [60, 91, 105] the goal of identifying factors motivating developers to perform refactoring operations, we adopt a different and complementary experimental design. First, we quantitatively investigate process- and product-related factors that may correlate with refactoring actions (e.g., does low code quality as assessed by quality metrics trigger refactoring operations?). Second, we qualitatively analyze refactorings performed by developers in the context of 551 merged PRs, to complement and validate the refactoring motivators already identified in the literature [60, 91, 105]. Note that, as for the work by Silva et al. [91] and differently from the ones by Wang et al. [105] and Kim et al. [60], we look at motivations related to specific refactoring operations (i.e., those performed in the analyzed PRs). Differently from Silva et al. [91], we do not rely on answers collected through a survey, but we inspected the code changes performed in the commits related to the PRs, and read the reviewing process carried out before merging the PR and the related discussion. As previously discussed, we validated and complemented the taxonomy of motivations they defined.

Peruma et al. [85] also look at the motivations pushing developers to refactor their code, but they focused only on rename refactoring operations, finding that, in most cases, renaming is applied to narrow the identifier meaning. Our qualitative study confirmed this finding.

Bavota et al. [39] study the relationship among metrics, code smells, and refactoring. Their findings indicate that there is no clear causation between code having a smell or exhibiting elevated complexity metric values and subsequent refactoring of this code. Our study, performed at the commit level, confirms their results but also points out that readability metrics and process-related factors play a significant role.

Finally, very related work in this line of research is the one by Vassallo et al. [104], in which the authors mine 200 systems to quantitatively investigate factors correlating with refactoring. They consider, in isolation, factors related to when, why, and by whom refactoring is performed. They found that (i) refactorings are mostly performed after one year from the project startup and rarely close to a new release, (ii) most of refactorings are performed while enhancing existing features, and (iii) developers that refactor code are often the owners of the impacted files. Our work, besides quantitatively investigating a more comprehensive set of process- and product-related factors (considered altogether in a mixed model), also complements this analysis with a qualitative investigation on the motivations behind refactorings.

6 CONCLUSION

In this article, we quantitatively and qualitatively analyzed the reasons behind refactoring operations performed by developers. Our quantitative analysis highlighted that (i) code readability is the product-related factor mostly correlated with refactoring operations and (ii) process-related factors such as source code change- and fault-proneness and, especially, the experience of developers changing a code component, play a significant role in triggering refactoring operations. Our qualitative analysis resulted in an extensive taxonomy of 67 motivations behind refactorings,

relating to quantitative results where possible. We have made the study material and data available in our replication package [27].

The implications of our study trigger several directions for future work.

Identifying "when" to trigger refactoring recommendations. Our quantitative study (RQ_1) shed some light on the product and process-related factors that contribute to trigger refactoring operations. Such an empirical evidence represents the basis for future approaches able to predict "when," during a project's evolution to trigger refactoring recommendations. Indeed, such an aspect is currently ignored in the refactoring recommenders literature, with researchers focusing their attention on the core problem of generating meaningful recommendations. However, recommending refactorings in a "context" in which developers do not feel in need to refactoring their code is unlikely to provide benefits. An interesting research direction in this field is to build on top of our RQ_1's findings to devise models able to predict when refactoring recommendations would be welcome by software developers.

On the relevance of "semantic metrics" for code smell detectors. As shown in our RQ_1, the product metrics exhibiting the stronger relationship with refactoring operations are those exploiting textual information extracted from the code, usually referred to as semantic metrics. This was the case for the readability and for the Conceptual Cohesion of Classes (C3) metrics. This result somehow supports the previous findings reported in the literature indicating a stronger alignment between the developers' perception of code quality and semantic metrics as compared to structural ones [37]. These findings suggest the adoption of these metrics in code smell detectors that, at the end, are used to identify refactoring opportunities.

On learning refactoring operations. While discussing the results of our qualitative study (RQ_2), a bold message repeatedly emerged from the analyzed cases: the motivations behind a refactoring are variegated and heterogeneous and, probably, cannot be captured by any combination of metrics. This implies that refactoring recommenders using product- and/or process-metrics as fitness functions to recommend refactorings will always miss many refactoring opportunities. Indeed, these techniques mostly target complex code components for refactoring (e.g., God Classes in the case of extract class refactoring), ignoring all the other scenarios we documented (e.g., applying extract class to remove a code clone). One possibility worth exploiting in the future is the application of deep learning techniques to refactoring recommenders. Indeed, recent work already shown the possibility to learn from code changes [102]. However, no previous work has attempted to design refactoring recommenders learning from developers' activities what a meaningful refactoring is in a given context.

Automatic support for relevant refactorings is missing. In RQ_2 we found many cases of manually performed refactoring operations that could benefit from the development of techniques to automate such code transformations. This includes the identification and automatic refactoring of redundant code and misuse of checked/unchecked exceptions in Java as well as the lack of support for preemptive refactoring (i.e., refactoring operations aimed at accommodating future changes). We see these areas as possible directions for future work in software refactoring.

Lack of production-ready tools. Some of the cases discussed in RQ_2 highlighted that, while the research community has developed many good approaches for supporting specific refactorings, these approaches find little application in practice. One clear example of this are the rename refactoring operations suggested by reviewers to expand abbreviations used in the identifiers of the code contributed in PRs. These recommendations could be easily generated at commit time by one of the many approaches proposed in the literature for the automatic expansion of abbreviations (see e.g., Reference [66]). However, the lack of production-ready tools could be the reason for the

lack of adoption of such techniques. This is an opportunity not only for researchers, but also for developers interested in building tools useful for the partial automation of code review activities.

On the need for pursuing "tradeoffs" when refactoring. Finally, our RQ$_2$ also highlighted the need for refactoring techniques able to consider the many contrasting objectives that a code transformation brings with it. Indeed, while most of the refactoring recommenders strive to maximize maintainability, our findings show that maintainability is only one of the many aspects considered by developers while refactoring, accompanied by performance, testability, etc. Future approaches to support refactoring should consider the pros and cons of the recommended solutions from different perspectives, suggesting operations that are sensible to different quality criteria.

Based on the above findings, our future research agenda will focus on the two points described above: (i) predicting *when* to recommend refactorings and (ii) developing refactoring tools sensible to different non-functional requirements.

REFERENCES

[1] [n.d.]. 52North/WPS Pull Request #73. Retrieved from https://github.com/52North/WPS/pull/73.
[2] [n.d.]. apache/fineract Pull Request #366. Retrieved from https://github.com/apache/fineract/pull/366.
[3] [n.d.]. cbeust/testng Pull Request #1481. Retrieved from https://github.com/cbeust/testng/pull/1481.
[4] [n.d.]. Code metrics for Java code by means of static analysis. Retrieved from https://github.com/mauricioaniche/ck.
[5] [n.d.]. confluentinc/kafka-connect-elasticsearch Pull Request #251. Retrieved from https://github.com/confluentinc/kafka-connect-elasticsearch/pull/251.
[6] [n.d.]. dropwizard/dropwizard Pull Request #488. Retrieved from https://github.com/dropwizard/dropwizard/pull/488.
[7] [n.d.]. DSpace/DSpace Pull Request #1083. Retrieved from https://github.com/DSpace/DSpace/pull/1083.
[8] [n.d.]. DSpace/DSpace Pull Request #324. Retrieved from https://github.com/DSpace/DSpace/pull/324.
[9] [n.d.]. eclipse/microprofile-fault-tolerance Pull Request #363. Retrieved from https://github.com/eclipse/microprofile-fault-tolerance/pull/363.
[10] [n.d.]. google/error-prone Pull Request #1071. Retrieved from https://github.com/google/error-prone/pull/1071.
[11] [n.d.]. kiegroup/optaplanner Pull Request #150. Retrieved from https://github.com/kiegroup/optaplanner/pull/150.
[12] [n.d.]. minio/minio-java Pull Request #238. Retrieved from https://github.com/minio/minio-java/pull/238.
[13] [n.d.]. PMD Source Code Analyzer. Retrieved from http://pmd.sourceforge.net.
[14] [n.d.]. samtools/htsjdk Pull Request #1067. Retrieved from https://github.com/samtools/htsjdk/pull/1067.
[15] [n.d.]. samtools/htsjdk Pull Request #599. Retrieved from https://github.com/samtools/htsjdk/pull/599.
[16] [n.d.]. spring-io/sagan Pull Request #179. Retrieved from https://github.com/spring-io/sagan/issues/179.
[17] [n.d.]. spring-io/sagan Pull Request #317. Retrieved from https://github.com/spring-io/sagan/pull/317.
[18] [n.d.]. spring-projects/spring-amqp Pull Request #346. Retrieved from https://github.com/spring-projects/spring-amqp/pull/346.
[19] [n.d.]. TeamAmaze/AmazeFileManager Pull Request #1577. Retrieved from https://github.com/TeamAmaze/AmazeFileManager/pull/1577.
[20] [n.d.]. zalando/nakadi Pull Request #626. Retrieved from https://github.com/zalando/nakadi/pull/626.
[21] [n.d.]. zalando/nakadi/ Pull Request #933. Retrieved from https://github.com/zalando/nakadi/pull/933.
[22] [n.d.]. zhcet-amu/zhcet-web Pull Request #136. Retrieved from https://github.com/zhcet-amu/zhcet-web/pull/136.
[23] H. Akaike. 1973. Information theory and an extension of the maximum likelihood principle. In *Proceedings of the 2nd International Symposium on Information Theory.*
[24] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14).* 281–293.
[25] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* 3 (January 2019), 40:1–40:29.
[26] Mohammad Alshayeb. 2009. Empirical investigation of refactoring effect on software quality. *Inf. Softw. Technol.* 51, 9 (2009), 1319–1326.
[27] Anonymous. 2018. Replication Package. Retrieved from https://github.com/replication-package/why-refactoring.
[28] Nicolas Anquetil and Timothy Lethbridge. 1999. Experiments with clustering as a software remodularization method. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'99).* 235–255.

[29] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM, 23.

[30] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. 2012. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA'12)*. 33–42.

[31] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. A new family of software anti-patterns: Linguistic anti-patterns. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. 187–196.

[32] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting linear mixed-effects models using lme4. *J. Stat. Softw.* 67, 1 (2015), 1–48.

[33] Gabriele Bavota. 2012. Using structural and semantic information to support software refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Computer Society, 1479–1482.

[34] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*. 104–113.

[35] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2013. Automating extract class refactoring: An improved method and its evaluation. *Empirical Software Engineering* 19, 6 (2013), 1–48.

[36] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*. Springer, Berlin, 387–419.

[37] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers' perception of software coupling. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE Computer Society, 692–701.

[38] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2013. Using structural and semantic measures to improve software modularization. *Empir. Softw. Eng.* 18, 5 (2013), 901–932.

[39] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.* 107 (2015), 1–14.

[40] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. 1998. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons.

[41] Raymond P. L. Buse and Westley Weimer. 2010. Learning a metric for code readability. *IEEE Trans. Softw. Eng.* 36, 4 (2010), 546–558.

[42] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. 2014. How changes affect software entropy: An empirical study. *Emp. Softw. Eng.* 19, 1 (2014), 1–38.

[43] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. 465–475.

[44] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. 74–83.

[45] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh, and I-Lang Wu. 2009. Exception handling refactorings: Directed by goals and driven by bug fixing. *J. Syst. Softw.* 82, 2 (2009), 333–345.

[46] Shyam R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20, 6 (June 1994), 476–493.

[47] Cesar Couto, João Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. 2011. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Softw. Qual. J.* 21, 2 (2011), 241–257.

[48] Florian Deissenbock and Markus Pizka. 2005. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*. 97–106.

[49] Danny Dig. 2011. A refactoring approach to parallelism. *IEEE Softw.* 28, 1 (2011), 17–22.

[50] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional.

[51] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola O. Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)*. 286–296.

[52] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*. 23.

[53] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 1037–1039.

[54] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA.

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

[56] Mathew Hall, Neil Walkinshaw, and Phil McMinn. 2012. Supervised software modularisation. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*. 472–481.

[57] Frank E. Harrell Jr, with contributions from Charles Dupont, et al. 2017. *Hmisc: Harrell Miscellaneous*. R package version 4.0-3. Retrieved from https://CRAN.R-project.org/package=Hmisc.

[58] Brian Henderson-Sellers, Larry L. Constantine, and Ian M. Graham. 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Obj. Orient. Syst.* 3 (1996), 143–158.

[59] David Kawrykow and Martin P. Robillard. 2009. Improving API usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. 111–122.

[60] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*.

[61] Miryung Kim, T. Zimmermann, and N. Nagappan. 2014. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.* 40, 7 (July 2014), 633–649.

[62] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first? In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07)*. 45–54.

[63] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 489–498.

[64] Giri P. Krishnan and Nikolaos Tsantalis. 2014. Unification and refactoring of clones. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. 104–113.

[65] A. Kuhn, S. Ducasse, and T. Gîrba. 2007. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.* 49, 3 (2007), 230–243.

[66] D. Lawrie and D. Binkley. 2011. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM'11)*. 113–122.

[67] Dawn Lawrie, Henry Feild, and David Binkley. 2006. Syntactic identifier conciseness and consistency. In *Proceedings of the Source Code Analysis and Manipulation Working Conference (SCAM'06)*. 139–148.

[68] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a name? a study of identifiers. In *Proceedings of the International Conference on Program Comprehension (ICPC'06)*.

[69] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innov. Syst. Softw. Eng.* 3, 4 (2007), 303–318.

[70] António Menezes Leitão. 2004. Detection of redundant code using R2D2. *Softw. Qual. J.* 12, 4 (1 December 2004), 361–382.

[71] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. 2017. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *Proceedings of the 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*. IEEE, 81–90.

[72] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are refactorings to blame? An empirical study of refactorings in merge conflicts. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER19)*. 151–162.

[73] Onaiza Maqbool and Haroon A. Babri. 2007. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.* 33, 11 (2007), 759–780.

[74] A. Marcus and D. Poshyvanyk. 2005. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 133–142.

[75] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw. Eng.* 34, 2 (2008), 287–300.

[76] John Micco. 2016. Flaky tests at google and how we mitigate them. Retrieved from https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[77] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.* 36, 1 (2010), 20–36.

[78] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2007. A case study on the impact of refactoring on quality and productivity in an Agile team. In *Balancing Agility and Formalism in Software Engineering, Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques (CEE-SET'07), Poznan, Poland, October 10-12, 2007, Revised Selected Papers*. 252–266.

[79] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. 181–190.

[80] Emerson Murphy-Hill, Chris Parnin, and Andreaw P. Black. 2011. How we refactor, and how we know it. *Trans. Softw. Eng.* 38, 1 (2011), 5–18.

[81] Matheus Paixão, Mark Harman, Yuanyuan Zhang, and Yijun Yu. 2018. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 394–414.

[82] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Emp. Softw. Eng.* 23, 3 (2018), 1188–1221.

[83] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. 2015. Anti-pattern detection: Methods, challenges, and open issues. *Adv. Comput.* 95 (2015), 201–238.

[84] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. 2018. Towards just-in-time refactoring recommenders. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)*. 312–315.

[85] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR'18)*. 26–33.

[86] Kata Praditwong, Mark Harman, and Xin Yao. 2011. Software module clustering as a multi-objective search problem. *IEEE Trans. Softw. Eng.* 37, 2 (2011), 264–282.

[87] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.

[88] B. Rosner. 2011. *Fundamentals of Biostatistics* (7th ed.). Brooks/Cole, Boston, MA.

[89] Chanchal K. Roy. [n.d.]. Large scale clone detection, analysis, and benchmarking: An evolutionary perspective. In *Proceedings of the 12th IEEE International Workshop on Software Clones (IWSC'18)*.

[90] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *J. Softw.: Evol. Process* 30, 6, e1958.

[91] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 858–870.

[92] Jaime Spacco, David Hovemeyer, and William Pugh. 2006. Tracking defect warnings across versions. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*. ACM, 133–136.

[93] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven code review: An empirical study. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 1061–1072.

[94] Donna Spencer. 2009. *Card Sorting: Designing Usable Categories*. Rosenfeld Media.

[95] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality (WoSQ'07)*. IEEE Computer Society, Los Alamitos, CA. http://dx.doi.org/10.1109/WOSQ.2007.11

[96] Gábor Szoke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*. IEEE, 95–104.

[97] Armstrong Takang, Penny Grubb, and Robert Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experiential study. *J. Program. Lang.* 4, 3 (1996), 143–167.

[98] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.* 35, 3 (2009), 347–367.

[99] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 483–494.

[100] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 60–70.

[101] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Softw. Eng.* 43, 11 (2017), 1063–1088.

[102] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 25–36.

[103] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP'01)*. 92–95.

[104] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Sci' Comput' Program'* 180, 1 (2019), 1–15.

[105] Yi Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'09)*. 413 –416.

[106] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. 2012. Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. 233–248.