An Exploratory Study on Refactoring Documentation in Issues Handling

Eman Abdullah AlOmar
Stevens Institute of Technology
Hoboken, New Jersey, USA
ealomar@stevens.edu

Anthony Peruma
Rochester Institute of Technology
Rochester, New York, USA
axp6201@rit.edu

Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, New York, USA
mwmvse@rit.edu

Christian D. Newman
Rochester Institute of Technology
Rochester, New York, USA
cnewman@se.rit.edu

Ali Ouni
ETS Montreal, University of Quebec
Montreal, Quebec, Canada
ali.ouni@etsmtl.ca

ABSTRACT

Understanding the practice of refactoring documentation is of paramount importance in academia and industry. Issue tracking systems are used by most software projects enabling developers, quality assurance, managers, and users to submit feature requests and other tasks such as bug fixing and code review. Although recent studies explored how to document refactoring in commit messages, little is known about how developers describe their refactoring needs in issues. In this study, we aim at exploring developer-reported refactoring changes in issues to better understand what developers consider to be problematic in their code and how they handle it. Our approach relies on text mining 45,477 refactoring-related issues and identifying refactoring patterns from a diverse corpus of 77 Java projects by investigating issues associated with 15,833 refactoring operations and developers' explicit refactoring intention. Our results show that (1) developers mostly use move refactoring related terms/phrases to target refactoring-related issues; and (2) developers tend to explicitly mention the improvement of specific quality attributes and focus on duplicate code removal. We envision our findings enabling tool builders to support developers with automated documentation of refactoring changes in issues.

CCS CONCEPTS

• **Software Engineering** → **Software Quality**; *Refactoring*.

KEYWORDS

Refactoring documentation, issues, software quality, mining software repositories

ACM Reference Format:
Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. 2022. An Exploratory Study on Refactoring Documentation in Issues Handling. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528525

1 INTRODUCTION

Code refactoring is a disciplined software engineering practice that is known as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [4, 14]. Refactoring is commonly used in different development and maintenance tasks [7]. It supports developers in revolving submitted issues such as feature requests [20] or bug reports [12, 15]. Issue tracking systems are used by most contemporary software projects enabling developers, quality assurance, managers, and users to submit feature or enhancement requests, as well as other tasks such as bug fixing and code review.

Previous studies have focused on recommending refactorings through the detection of refactoring opportunities, either by identifying code anti-patterns that need correction [10, 21], or by optimizing code quality metrics [9, 17]. Yet, recent studies have shown that there is a gap between automated refactoring tools, and what developers consider to a need-to-refactor situation in code [11, 13]. To bridge this gap, it is important to understand what triggers developers to refactor their code, and what do developers care about when it comes to code improvement. Such information provides insights, to software practitioners and researchers, about the developer's perception of refactoring. This can question whether developers do care about structural metrics and code smells when refactoring their code, or if there are other factors that are of direct influence on these non-functional changes.

In this paper, we focus on investigating issues that are written to express a need for refactoring. We extract patterns differentiating refactoring issues. These patterns represent what developers consider to be worth refactoring. We are also interested in the solutions, *i.e.*, refactoring operations, being proposed as correction measures. Our investigation is driven by answering the following research questions:

- **RQ$_1$:** ***What textual patterns do developers use to describe their refactoring needs in issues?*** This RQ explores the existence of refactoring documentation in issues containing refactorings. This RQ aims to identify developers' common phrases when describing their refactoring problem/challenge.
- **RQ$_2$:** ***What are the quality attributes developers care about when documenting in issues?*** In this RQ, we investigate whether developers explicitly indicate the purpose of

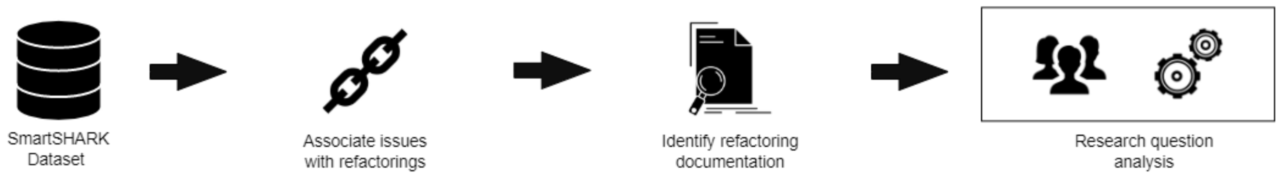
SmartSHARK Dataset
Associate issues with refactorings
Identify refactoring documentation
Research question analysis

Figure 1: Overview of our experiment design.

their refactoring activity applied in issues, *e.g.,* improving structural metrics of fixing code smells.

The results of this exploratory study strengthen our understanding of what circumstances cause the need for refactorings. Using the evolution history of 77 open-source projects exhibiting a total of 45,477 refactoring commits with issues, our study reveals that developers are mainly driven by reducing complexity and increasing comprehension and performance. While various studies associate refactoring tightly with fixing code smells, the only anti-pattern that was highlighted is duplicate code. Furthermore, various studies have shown that the *rename* category is the most frequent in terms of refactoring operations (*e.g.*, rename method, rename attribute, etc.) [19, 23], but our study reveals that developers tend also to discuss more complex refactorings, including refactorings dealing with *extract* and *move* code fragments, due to their impact on the design and code semantics preservation. We have also prepared a replication package of issues and their corresponding fixes (refactorings), to support the reproducibility and extension of our work [1].

2 STUDY DESIGN

Figure 1 depicts a general overview of our experimental setup. In the following subsections, we elaborate on the activities involved in the process. We provide the dataset that we generate available in our replication package [1].

2.1 Source Dataset

Our study utilizes the SmartSHARK MongoDB Release 2.1 dataset [29]. This dataset contains a wide range of information for 77 open-source Java projects, such as commit history, issues, refactorings, code metrics, mailing lists, and continuous integration data, among others. All 77 Java projects are part of the Apache ecosystem and utilize GitHub as their version control repository and JIRA for issue tracking. Furthermore, SmartSHARK utilizes RefDiff [27] and RefactoringMiner [32] to mine refactoring operations. Finally, the SmartSHARK dataset schema provides the necessary relationship attributes between data collections to join two or more related collection types.

2.2 Refactoring Documentation Detection

To identify refactoring documentation patterns in issues, we perform a series of manual and automated activities, as follows:

Step #1: Issues associated with a refactoring activity. As our study focuses on issues and refactorings, our analysis is limited to issues where one or more refactoring operations were performed

Table 1: Dataset overview of refactoring-related issues.

Item	Count
Refactoring commits with issues	45,477
Refactoring commits with issues having keyword '*refactor**' in title	835
Refactoring operations associated with issues	15,833
Issue Status	
Item	**Count**
Closed	603
In progress	8
Open	28
Resolved	196
Issue Resolution	
Item	**Count**
Done	4
Fixed	780
Implemented	8
Resolved	4
Won't fix	3
Issue Types	
Item	**Count**
Bug	95
Improvement	390
New Feature	2
Story	1
Sub-task	71
Task	274
Test	2

as part of the issue resolution. Hence, we first extracted all different refactorings from the source dataset. Next, we identify all commits containing the refactoring operations. Finally, we extracted issues that was addressed using the identified commits.

Step #2: Issues associated with developer intention about refactoring. To ensure that the selected issues are about refactoring, we focused on a subset of 835 issues that reported developers' intention about the application of refactoring (*i.e.*, having the keyword '*refactor*'). The choice of '*refactor*', besides being used by all related studies, is intuitively the first term to identify ideal refactoring-related issues [2, 5, 18]. Finally, to reduce the occurrence of false positives, we limited our analysis to only the occurrence of the term *'refactor'* in the title of the issue as the title is a concise description of the problem [24, 25].

Step #3: Annotation of issues. When creating issues, developers use natural language to describe the issues. Hence, given the diverse nature of developers describing the problem, an automated approach to analyzing the issue text is not feasible. Therefore, we performed a manual analysis of the issue title and body to identify refactoring documentation patterns. Next, we grouped this subset of issues based on specific patterns. Further, to avoid redundancy of any pattern, we only considered one phrase if we found different patterns with the same meaning. For example, if we find patterns such as 'simplifying the code', 'code simplification', and 'simplify code', we add only one of these similar phrases in the list of patterns. This enables having a list of the most insightful and unique

Table 2: List of refactoring documentation in issues ('*' captures the extension of the keyword) .

Patterns		
Add*	Chang*	Chang* the name
Clean* up code	Cleanup	Code clean*
Code optimization	Creat*	Extend*
Extract*	Fix*	Fix* code style
Improv*	Improv* code quality	Inlin*
Introduc*	Merg*	Mov*
Pull* up	Push* down	Repackag*
Redesign*	Reduc*	Refactor*
Refin*	Remov*	Renam*
Reorganiz*	Replac*	Restructur*
Rewrit*	Simplify* code	Split*

Table 3: Summary of refactoring patterns, clustered by refactoring related categories.

Internal QA (%)	External QA (%)	Code Smell (%)
Complexity (0.26 %)	Readability (0.23 %)	Duplicate code (0.73 %)
Design Size (0.25 %)	Performance (0.11 %)	
Encapsulation (0.18 %)	Usability (0.08 %)	
Dependency (0.16 %)	Extensibility (0.07 %)	
Inheritance (0.08 %)	Compatibility (0.06 %)	
Coupling (0.02 %)	Accuracy (0.05 %)	
Abstraction (0.02 %)	Modularity (0.05 %)	
	Flexibility (0.04 %)	
	Understandability (0.04 %)	
	Reusability (0.04 %)	
	Testability (0.03 %)	
	Maintainability (0.03 %)	
	Manageability (0.02 %)	
	Stability (0.006 %)	
	Accessibility (0.01 %)	
	Configurability (0.01 %)	
	Robustness (0.006 %)	
	Repeatability (0.006 %)	
	Effectiveness (0.006 %)	

patterns, and it also helps in making more concise patterns that are usable for readers.

3 EXPERIMENTAL RESULTS

3.1 RQ$_1$: What textual patterns do developers use to describe their refactoring needs in issues?

Methodology. To identify refactoring documentation patterns, we manually inspect a subset of issues. These patterns are represented in the form of a keyword or phrase that frequently occurs in the issues associated with refactoring-related commits.

Results. Our in-depth inspection of the issues results in a list of 33 refactoring documentation patterns, as shown in Table 2. Our findings show that the names of refactoring operations (*e.g.*, 'extract*', 'mov*', 'renam*') occur in the top frequently occurring patterns, and these patterns are mainly linked to code elements at different levels of granularity such as classes, methods, and variables. These specific terms are well-known software refactoring operations and indicate developers' knowledge of the catalog of refactoring operations. We also observe that the top-ranked refactoring operation-related keywords include 'extract*' and 'mov*'. 'pull up' and 'push down' operations are among the least discussed refactoring operations (similar to findings in [24, 26]). Moreover, we observe the occurrences of issue-fixing specific terms such as 'fix*', 'remov*', and 'reduc*'. Next, we examine the most common keywords that developers use when expressing refactoring documentation in issues. Figure 3 shows the top keywords used to identify refactoring documentations across the examined projects, that are ranked according to their number of occurrences.

To better understand the nature of refactoring documentation, we have classified the associated refactoring operations into 5 classes, namely, 'changing', 'extracting', 'inlining', 'moving', and 'renaming'. Depicted in Figure 2, we cluster these operations associated with issues using a list of refactoring keywords defined in a previous work [3]. The changing of the types belongs to the 'changing' class, whereas the extraction of classes and methods are included in the 'extracting' class. As for, 'moving', it gathers all the movement of code elements, *e.g.*, moving methods, or pushing code elements across hierarchies. Merging-related activities are included in the 'inlining' class. Finally, the 'renaming' class contains all refactorings that rename a given code element such as a class, a package or an attribute. As shown in Figure 2, the 'extracting' operations are highly documented in issues across the projects, and it reached the percentage of 37.2%, higher than 'moving', 'renaming', and 'changing' whose percentage is respectively 24.6%, 21.5%, and 14.7%. The 'inlining' operations, however, is the least documented refactoring which had a ratio of only 2 %.

3.2 RQ$_2$: What are the quality attributes developers care about when documenting in issues?

Methodology. After identifying the different refactoring documentation patterns, we identify and categorize the patterns into three main categories (similar to previous studies [5–8]): (1) internal quality attributes, (2) external quality attributes, and (3) code smells.

Results. Table 3 provides the list of refactoring documentation patterns, ranked based on their frequency, we identify in refactoring-related issues. We observe that developers frequently mention key internal quality attributes (such as inheritance, complexity, etc.), a wide range of external quality attributes (such as readability and performance), and code duplication code smell that might impact code quality. To improve the internal design, the system structure optimization regarding its complexity and design size seems to be the dominant focus that is consistently mentioned in issues (0.26% and 0.25%, respectively). Concerning external quality attribute-related issues, we observe the mention of refactorings to enhance nonfunctional attributes. Patterns such as 'readability', 'performance', and 'usability' represent the developers' main focus, with 0.23%, 0.11%, and 0.08%, respectively. Finally, for code smell-focused refactoring issues, duplicate code represents the most popular anti-pattern developers intend to refactor (0.73%).

4 DISCUSSION

Our research aims to explore refactoring documentation in issues to provide future research directions that support developers in understanding refactoring applied in issues.

RQ$_1$ indicates that developers tend to use a variety of textual patterns to document their refactorings in issues. These patterns can provide either a (1) generic description of problems developers

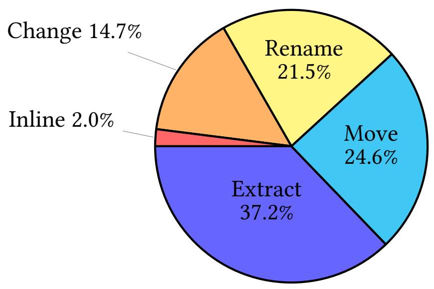

Figure 2: Percentage of refactorings, clustered by operation class.

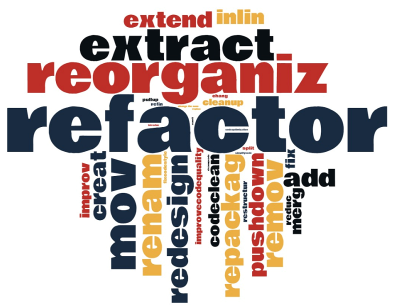

Figure 3: Popular refactoring textual patterns in issues.

encounter or (2) a specific refactoring operation name following Fowler's names [14]. Although previous studies show that rename refactorings are a common type of refactoring, *e.g.*, [23], we notice that 'mov*' and 'extract*' are the topmost documented refactorings in issues. This can be explained by the fact that developers tend to make many design improvement decisions that include remodularizing packages by moving classes, reducing class-level coupling, and increasing cohesion by moving methods. Additionally, developers might use similar terminology when performing move-related refactoring operations, *i.e.*, *Extract/Inline/Pull-up/Push-down* [3]. As shown in Figure 2, 'extracting' is the most documented refactorings. An interpretation for this comes from the nature of the debugging process that may include the separation of concerns which helps in reducing the core complexity of a larger module and reduce its proneness to errors [31]. In other words, developers tend to discuss more complex refactorings in issues, including refactorings from the *extract* and *move* categories, due to their impact on the design and code semantic preservation. This information can provide valuable references for refactoring documentation practice in issues. For example, whether refactoring-related issue descriptions have the relevant information is a critical indicator for reproducing refactoring-related issues.

From **RQ**$_2$, we observe that developers discuss quality concerns when documenting refactorings in issues that can be related to: (1) internal quality attributes, (2) external quality attributes, or (3) code smells. When analyzing these quality concerns per issue types reported in Table 1, we notice that complexity and duplicate code are mostly documented with issue type named 'bug', whereas the duplicate code and readability were the popular sub-categories for refactoring-related issues type named 'improvement'. For instance, the developer discussed fixing design issues by putting common functionalities into a superclass to eliminate duplicate code, breaking up lengthier methods to make the code more readable, and avoiding nested complex data structure to reduce code complexity. Moreover, we observe that code smell is rarely documented in issues. As shown in Table 3, developers only focused on duplicate code removal. Conversely, developers tend to report a variety of external quality attributes, focusing mainly on improving *readability* of the code. This corroborates the finding by Palomba et al. [22], where refactoring targeting program comprehension was mostly applied during bug fixing activities. As developers discussed functional and non-functional aspects of source code, future research can further investigate the intent as to why and how developers perform refactoring in issues. With a better understanding of this phenomenon, researchers and tool builders can support developers with automatically documenting refactorings in issues.

One of the main purposes of exploring refactoring documentation in issues is to better understand how developers cope with their software decay by extracting any refactoring strategies that can be associated with removing code smells [30], or improving the design structural measurements [16]. However, these techniques only analyze the changes at the source code level, and provide the operations performed, without associating it with any textual description, which may infer the rationale behind the refactoring application. Our proposal, of textual patterns, is the first step towards complementing the existing effort in detecting refactorings, by augmenting it with any description that was intended to describe the refactoring activity. As previously shown in Tables 2 and 3, developers tend to add a high-level description of their refactoring activity, and mention their intention behind refactoring (remove duplicate code, improve readability, etc.), along with mentioning the refactoring operations they apply (type migration, inline methods, etc.).

Overall, the documentation of refactoring in issues is an important research direction that requires further attention. It has been known that there is a general shortage of refactoring documentation, and there is no consensus about how refactoring should be documented, which makes it subjective and developer-specific. Lack of design documentation forced developers to rely on the source code to identify design problems [28]. Moreover, the fine-grained description of refactoring can be time-consuming, as a typical description should contain an indication about the operations performed, refactored code elements, and a hint about the intention behind the refactoring. In addition, the developer specification can be ambiguous as it reflects the developer's understanding of what has been improved in the source code, which can be different in reality, as the developer may not necessarily adequately estimate the refactoring impact on the quality improvement.

5 THREATS TO VALIDITY

The first threat relates to the analysis of open-source Java projects. Our results may not generalize to systems written in other languages. Another potential threat to validity relates to our findings regarding counting the reported quality attributes and code smells. Due to the large number of issues associated with refactorings, we have not performed a manual validation to remove false positive issues. Thus, this may have an impact on our findings. Finally, we constructed our dataset by extracting issues containing the term 'refactor' in the title. There is the possibility that we may have excluded synonymous terms/phrases. However, even though this approach reduces the number of issues in our dataset, it also decreases false-positives, and ensures that we analyze issues that are explicitly focused on refactorings.

6 CONCLUSION & FUTURE WORK

In this study, we performed an exploratory study to understand how developers document refactorings in issues. Specifically, we identify refactoring terms/phrases patterns, study possible refactoring documentation types, and determine how many refactoring terms/phrases exist in issues. Our results show that (1) developers mostly use move refactoring related terms/phrases to target refactoring-related issues; and (2) developers tend to explicitly mention the improvement of specific quality attributes and focus on duplicate code removal. We envision our findings enabling tool builders to support developers with automatically document refactoring in issues. Future work in this area includes investigating which refactoring operation is more problematic in issues.

REFERENCES

[1] AlOmar. 2022. *ReplicationPackage.* https://smilevo.github.io/self-affirmed-refactoring/

[2] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 348–357.

[3] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Automated Software Engineering* 29, 1 (2022), 1–40.

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* (2021), 106675.

[5] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *International Workshop on Refactoring-accepted. IEEE.*

[6] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2021. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software* 171 (2021), 110821.

[7] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. 2021. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications* 167 (2021), 114176.

[8] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. 2021. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process* (2021), e2395.

[9] Sajid Anwer, Ahmad Adbellatif, Mohammad Alshayeb, and Muhammad Shakeel Anjum. 2017. Effect of coupling on software faults: An empirical study. In *2017 International Conference on Communication, Computing and Digital Systems (C-CODE).* IEEE, 211–215.

[10] Aline Brito, Andre Hora, and Marco Tulio Valente. 2020. Refactoring Graphs: Assessing Refactoring over Time. *arXiv preprint arXiv:2003.04666* (2020).

[11] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering.* ACM, 73–82.

[12] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. 2020. On the relationship between refactoring actions and bugs: a differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 556–567.

[13] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology* 126 (2020), 106347.

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. http://dl.acm.org/citation.cfm?id=311424

[15] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are SonarQube Rules Inducing Bugs?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 501–511.

[16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* ACM, 331–336.

[17] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–45.

[18] Emerson Murphy-Hill, Andrew P Black, Danny Dig, and Chris Parnin. 2008. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools.* 1–5.

[19] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming.* Springer, 552–576.

[20] Ally S Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. 2019. Automated recommendation of software refactorings based on feature requests. In *2019 IEEE 27th International Requirements Engineering Conference (RE).* IEEE, 187–198.

[21] Willian Oizumi, Ana C Bibiano, Diego Cedrim, Anderson Oliveira, Leonardo Sousa, Alessandro Garcia, and Daniel Oliveira. 2020. Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. In *Proceedings of the 34th Brazilian Symposium on Software Engineering.* 72–81.

[22] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC).* IEEE, 176–185.

[23] 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.* 26–33.

[24] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do I refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering* 27, 1 (2022), 1–43.

[25] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 1192–1223. https://doi.org/10.1007/s10664-015-9379-3

[26] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016).* ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[27] Danilo Silva and Marco Tulio Valente. 2017. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 269–279.

[28] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, et al. 2018. Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering.* 921–931.

[29] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. 2021. MSR Mining Challenge: The SmartSHARK Repository Data. (2021).

[30] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering.* IEEE, 329–331.

[31] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

[32] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 483–494.