Towards Better Understanding Developer Perception of Refactoring

Eman Abdullah AlOmar
Rochester Institute of Technology, NY, USA
eman.alomar@mail.rit.edu

***Abstract*—Refactoring is a critical task in software maintenance. It is typically performed to enforce best design practices, or to cope with design defects. Research in refactoring has been driven by the need to improve system structures. However, recent studies have shown that developers may incorporate refactoring strategies in other development-related activities that go beyond improving the design. Unfortunately, these studies are limited to developer interviews and a reduced set of projects. In this context, we aim at exploring how developers document their refactoring activities during the software life cycle, we call such activity Self-Affirmed Refactoring (SAR), by understanding developers perception of refactorings so that we can bridge the gap between refactoring and automation in general. We aim in more accurately mimicking the human decision making when recommending better software refactoring and remodularization.**

I. Introduction

The success of a software system depends on its ability to retain a high quality of design in the face of continuous change. However, managing the growth of the software while continuously developing its functionalities is challenging, and can account for up to 75% of the total development cost. One key practice to cope with this challenge is refactoring. Refactoring is the art of remodeling the software design without altering its functionalities [7], [9]. It was popularized by [9], who identified 72 refactoring types and provided examples of how to apply them in his catalog.

Refactoring is the art of improving the quality of software design without altering its behavior. With the rise of agile methodologies that encourage developers to interleave refactoring within other development activities, and with the incorporation of refactoring operations in modern Integrated Development Environments (IDEs), there is a lot of growing research to better understand how developers practically refactor their codebases [17], [18], [20]. Thus, several studies focus on detecting refactoring operations, performed by developers, by mining their commit changes and extracting the refactoring history [12], [24], [26]. These refactoring detectors rely mainly on analyzing code changes to identify refactorings strategies, previously performed by developers in various development contexts.

In order to learn from these refactoring strategies, it is essential to also understand the developer's rationale and intent behind applying them, *i.e.*, the context in which the refactoring operations were executed. Existing studies on understanding developers perception of refactorings mainly rely on developers surveys and formal interviews [10], [13]. As the existing refactoring detectors offer an abundant source of commits containing refactoring operations, this paper aims at exploring how developers document their refactoring activities during the software life-cycle.

Inspired by various studies in analyzing the developer's internal documentation to extract their perception of their own code, *e.g.*, self-admitted technical debt [6], [21], we text-mine the developer's messages in refactoring-related commits to detect any potentially relevant information regarding the applied refactorings. Indeed, commit messages represent an atomic documentation of a code change, written by the change author, and thus represents a reliable and rich source of information to describe their intention behind the performed changes. Therefore, we conduct this empirical study to identify how developers describe their refactoring activities. Then we extract the rationale behind the applied refactorings, *e.g.*, fixing code smells or improving specific quality attributes.

The purpose of this thesis research is to extend further our basic definition of refactoring and to challenge the existing case studies; advocating a need for research on refactorings that takes into account the fact that refactorings are not always solicited in the same context of improving the internal design by fixing smells. Our goal is to *extract* the developer's perception of refactoring by *mining* its refactoring operations, and positioning their context of usage within the development process, in order to better simulate the human decision making when *recommending* refactorings to improve software design. To achieve this goal, this thesis proposes the following contributions:

Research Thrust 1. Mining-to-define the developer's taxonomy of refactoring documentation. As a first contribution, we extract the refactoring *knowledge*, by mining code instances containing refactoring related operations. Then, we define the taxonomy associated with how developers document their code optimization. This first research thrust has resulted in our first contribution which was recently accepted in the 3rd International Workshop on Refactoring, co-located with the International Conference on Software Engineering (IWoR@ICSE) [3]. It was also awarded the **best paper award**, and the **best presentation award**.

Research Thrust 2. Mining developer perception of refactoring. In this phase, we analyze the contexts in which developers are performing refactorings to better understand how developers interleave code optimization with other developments tasks. As we build towards answering this research

thrust, our second contribution was recently accepted in the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) [4].

Research Thrust 3. Supporting the automation of refactoring recommendation. By capturing real-world scenarios of refactorings when extracting refactoring patterns (RT 1), and understanding how they fit in software development contexts (RT 2) we can automate the refactoring process, not only by using state-of-art features (improving design metrics and quality attributes) but also with *contextual* features which better simulate the human behavior when facing software decay.

In this paper, we enumerate the research questions we are planning on addressing as part of exploring our research thrusts. Then we showcase the preliminary findings of our first contribution related to defining a taxonomy for refactoring documentation.

II. Literature Review

A number of studies have focused recently on the identification and detection of refactoring activities during the software life-cycle. One of the common approaches to identify refactoring activities is to analyze the commit messages in versioned repositories. Stroggylos & Spinellis [27] opted for searching words stemming from the verb *"refactor"* such as "refactoring" or "refactored" to identify refactoring-related commits. Ratzinger et al. [22], [23] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions to identify whether a transformation contains refactoring. The authors identified refactorings based on a set of keywords detected in commit messages.

Later, Murphy-Hill et al. [18] replicated Ratzinger's experiment in two open source systems using Ratzinger's 13 keywords. They conclude that commit messages in version histories are unreliable indicators of refactoring activities. This is due to the fact that developers do not consistently report/document refactoring activities in the commit messages. In another study, Soares et al. [26] compared and evaluated three approaches, namely, manual analysis, commit message (Ratzinger et al.'s approach [22], [23]), and dynamic analysis (SafeRefactor approach [25]) to analyze refactorings in open source repositories, in terms of behavioral preservation. The authors found that manual analysis achieves the best results in this comparative study and is considered as the most reliable approach in detecting behavior-preserving transformations.

In another study, Kim et al. [13] surveyed 328 professional software engineers at Microsoft to investigate when and how they do refactoring. They first identified refactoring branches and then asked developers about the keywords that are usually used to mark refactoring events in change commit messages. When surveyed, the developers mentioned several keywords to mark refactoring activities. Kim et al. matched the top ten refactoring-related keywords identified from the survey against the commit messages to identify refactoring commits from version histories. Using this approach, they found 94.29% of commits do not have any of the keywords, and only 5.76% of commits included refactoring-related keywords.

More recently, Zhang et al. [29] performed a preliminary investigation of Self-Admitted Refactoring (SAR) in three open source systems. They first extracted 22 keywords from a list of refactoring operations defined in Martin Fowler's book [9] as a basis for SAR identification. After identifying candidate SARs, they used Ref-Finder [12] to validate whether refactorings have been applied. In their work, they used code smells to assess the impact of SAR on the structural quality of the source code. Their main findings are the following (1) SAR tend to enhance the software quality although there is a small percentage of SAR that have introduced code smells, and (2) the most frequent code smells that are introduced or reduced depend highly on the nature of the studied projects. They concluded that SAR is a signal that helps to find refactoring events, but it does not guarantee the application of refactorings.

III. Research Plan

A. Research agenda

Figure 1 depicts the overview of the thesis. We want to follow the following research plan:

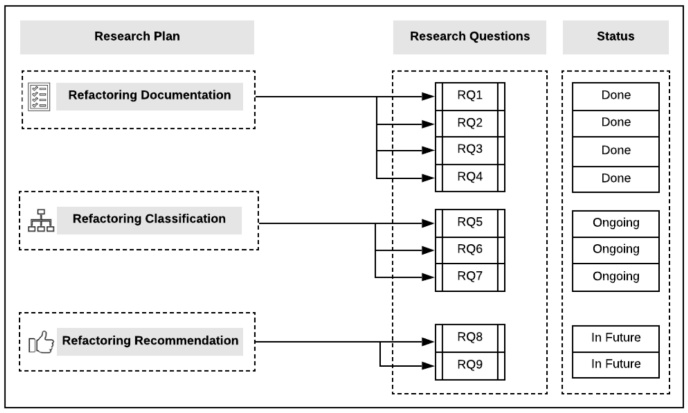

Fig. 1. Thesis Overview

1) ***Refactoring Documentation****:* In this phase, we analyze the developer's internal documentations of refactorings to introduce refactoring patterns. We answer the following research questions:

RQ1. *What patterns do developers use to describe their refactoring activities?* Since there is no consensus on how to formally document the act of refactoring code, we mine in this research question, patterns, using which developers have described their refactoring activities. We explore 322,479 commit messages, belonging to a large variety of projects. The outcome of this research question enumerates the most popular text patterns used in the analyzed commit messages.

RQ2. *What are the quality issues that drive developers to refactor?* Various studies have explored the bad programming practices that trigger refactoring and the potential quality attributes that are optimized when restructuring the code. In this research question, we investigate whether developers

explicitly mention the purpose of their refactoring activity, *e.g.,* improving structural metrics of fixing code smells.

RQ3. *Do commits containing the label "Refactor" indicate more refactoring activity than those without the label?* We revisit the hypothesis raised by Murphy-Hill et al. [17] about whether developers use a specific pattern, *i.e., "refactor"* when describing their refactoring activities. We quantify the messages with the label *"refactor"* and without to compare between them.

2) ***Refactoring Documentation Assessment****:* In this phase: we investigated the impact of SAR on quality by focusing on the main internal quality attributes and using well-known quality metrics reported in the literature. We specifically answer the following research question:

RQ4. *Do the developer perception of quality improvement align with the quantitative assessment of code quality?* Our main goal is to investigate whether the developer perception of quality improvement (as expected by developers) aligns with the real quality improvement (as assessed by quality metrics).

3) ***Refactoring Developers Perception & Practices****:* In this phase, we extend further our basic definition of refactoring by identifying the developers motivation behind every application of a refactoring targeting an answer to the following research questions:

RQ5. *What is the developer's purpose of the applied refactorings?* We determine the motivation of the developer through the classification of the commit containing the refactoring operations. It identifies the type of development tasks that refactorings were interleaved with, e.g., updating a feature, or debugging.

RQ6. *Is there a subset of developers, within the development team, who are responsible for refactoring the code?* This research question challenges the scalability of several papers that have performed developer interviews and demonstrated that only a subset of developers perform major refactoring activities. In our study, we verify whether the task of refactoring is equally distributed among all developers or if it is the responsibility of a subset of developers.

RQ7. *Are there more refactoring activities before project releases than after?* Refactoring is known to be solicited around major releases, yet there is not enough information on the nature of activities applied. To answer this research question, we monitor the refactoring activities for a time window centered by the release date, then we compare the frequency of refactoring before and after the release of the new version.

4) ***Refactoring Automation****:* In this phase, we plan on proposing an automated strategy to remodularize software packages by recommending the right package for a given class. We want also to make sure that our recommendation provides the optimum modular design of the software by improving the dominant modularization forces (*i.e,* cohesion and coupling). Therefore, we are seeking for an answer to the following research questions:

RQ8. *How to recommend the right package for a given class?* We plan to use a set of features and metrics, use learning to rank technique to rank the selected features, and apply clustering techniques to group similar packages together.

RQ9. *What is the impact of package-level refactorings on software quality?* We plan to assess the impact of four refactorings applied to code elements with higher granularity level, *i.e.*, move, merge, split, and rename, on software quality.

B. Methodology

1) Data Collection & Refactoring Detection: To perform our experimental study, we utilize an existing benchmark of GitHub repositories by Allamanis [2]. To extract the entire refactoring history for each project, we use the Refactoring Miner tool, developed by Tsantalis et al. [28]. Refactoring Miner is designed to analyze code changes in git repositories to detect applied refactoring. Our choice to use Refactoring Miner is justified by the fact that it achieved accurate results in detecting refactorings compared to the state-of-the-art available tools, with a precision of 98% and recall of 87% [24], [28]. In this phase, Refactoring Miner detected 1,208,970 refactoring operations in 3,795 projects.

2) Self-Affirmed Refactoring Analysis: After extracting all refactoring commit messages detected by Refactoring Miner, our next step consists of analyzing each of the commit messages. As for pattern identification, we were inspired by the manual analysis of Potdar and Shihab [21] when analyzing comments containing self-admitted technical debt. Similarly, since commit messages are written in natural language and we need to understand how developers express refactoring, we manually analyzed commit messages by reading through each message to identify SAR patterns. We then extracted these commit comments to specific patterns. To avoid redundancy of any kind of patterns, we only considered one phrase if we found different forms of patterns that have the same meaning. This enables us to have a list of the most insightful and unique patterns. It also helps in making more concise patterns that are usable for readers. The manual analysis process took approximately 7 days in total.

3) Selection of Quality Metrics: To set up a comprehensive set of quality metrics, we first conducted a literature review on existing and commonly above acknowledged software quality metrics [5], [8], [11], [15], [16], [19]. Then, we checked if the metrics assess several object-oriented design aspects in order to map each metric to the appropriate internal quality attributes. For example, the RFC (Response For Class) metric is typically used to measure the coupling and complexity quality attributes. Thereafter, we create a list of metrics and associate each of the well-known metrics (*e.g.,* CK suite [5] and McCabe [16]) with common quality attributes. The process left us with 19 object-oriented metrics. All metrics values are automatically computed using the tool UNDERSTAND[1], a software quality assurance framework.

[1]https://scitools.com/

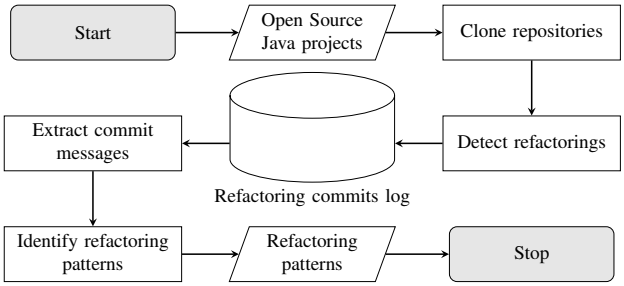

Fig. 2. Approach Overview.

C. Results

This section reports our experimental results described in [3], [4]. The dataset of refactorings is available online[2].

RQ1. *What patterns do developers use to describe their refactoring activities?* To identify SAR patterns, we manually inspect a subset of commit messages and categorize these messages into lexically or semantically similar patterns. These patterns are represented in the form of a keyword or phrase that frequently occur in the comments of all refactoring-related commits. The extraction of our approach has been carried through few iterations. We start our first iteration by searching for the term *"refactor*"*. In this iteration, we obtained 33,301 refactoring commit messages. Then, we started a manual inspection of each commit message that is associated with the term *"refactor"* to the set of patterns that are also used to describe the refactoring activity. As developers may not always use the term *"refactor"* explicitly to document their refactoring activities in their commit messages. Thus, to alleviate this issue, we reiterate again, using the extracted patterns in the first iteration, while excluding the term *"refactor"*, to identify additional SAR patterns. We kept iterating by extracting new patterns while excluding the previously identified ones until we are no longer able to find any relevant patterns. Our in-depth inspection resulted into a list of 87 SAR patterns identified across the considered projects.

RQ2. *What are the quality issues that drive developers to refactor?* We identify and categorize SAR patterns used to describe the motivation behind refactorings to three main categories: (1) internal quality attributes, (2) external quality attributes, and (3) code smells. We perform five sequential steps to answer this research question. We start by collecting software issues ,*i.e.,* quality attributes and code smells reported in the literature [1], [9], [14]. Then, we search for common categories among the reported quality attributes and code smells. The following step involves identifying categories clustering quality attributes under the identified categories. This process resulted in three different categories mentioned above. For each of the collected quality attributes and code smells, we search in our database for any potential refactoring commit that contains any of the collected quality attributes and code smells.

[2]https://smilevo.github.io/self-affirmed-refactoring/

RQ3. *Do commits containing the label "Refactor" indicate more refactoring activity than those without the label?* In an empirical context, we test Murphy-Hill et al.'s [17] hypothesis mentioned above in two rounds. In the first round, we used the keyword *"refactor"*, exactly as dictated by the authors. Thereafter, we quantified the proportion of commits including the searched label across all the considered projects in our benchmark. In the second round, we re-tested the hypothesis using the 87 SAR patterns we identified, *i.e.,* we counted the percentage of commits containing any of our SAR labels. The result of the two rounds resides in a strict set of commits containing the label *refactor*, which is included in a larger set containing all patterns, and finally a remaining set of commits which does not contain any patterns. For each of the sets, we count the number of refactoring operations identified in the commits. By comparing the different commits that are labeled and unlabeled with SAR patterns, we observe a significant number of labeled refactoring commits for each refactoring operation supported by the tool Refactoring Miner. This implies that there is a strong trend of developers in using these phrases in refactoring commits. The results for commits labeled and unlabeled *"refactor"* with engendering an opposite observation, which corroborates the expected outcome of Murphy-Hill et al.'s hypothesis. Thus, the use of *"refactor"* is not a great indication of refactoring activities. The difference between the two tests indicates the usefulness of the list of SAR patterns that we identified.

RQ4. *Do the developer perception of quality improvement align with the quantitative assessment of code quality?* A variety of structural metrics can represent the internal quality attributes considered in this study. Based on our empirical investigation, for metrics that are associated with quality attributes, there are different degrees of improvement and degradation of software quality. Additionally, most of the metrics that are mapped to the main quality attributes, i.e., cohesion, coupling, and complexity, do capture developer intentions of quality improvement reported in the commit messages. In contrast, there is also a case in which the metrics do not capture quality improvement as perceived by developers.

IV. Conclusion

In this paper, we explored how developers explicitly report refactoring activities in the commit messages of versioned repositories. We introduced 87 SAR patterns, which is an indication of the developer-reported refactoring events in the change messages, to capture developers refactoring taxonomy. We progressed on this direction by assessing the impact of SAR on quality and exploring several motivation behind refactorings that go beyond the traditional definition of refactorings. Therefore, our next plan is mainly focused on automatically remodularizing software packages while improving the dominant modularization driving forces, and empirically assessing the impact of package-level refactorings on quality.

V. Acknowledgments

The author would like to thank her advisor Mohamed Wiem Mkaouer for all the help and support received during the PhD.

References

[1] J. Al Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[2] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[3] E. A. AlOmar, M. W. Mkaouer, and A. Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3nd International Workshop on Refactoring*. IEEE, 2019.

[4] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[6] E. da Silva Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[7] J. A. Dallal and A. Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[8] G. Destefanis, S. Counsell, G. Concas, and R. Tonelli. Agile processes in software engineering and extreme programming. chapter Software Metrics in Agile Software: An Empirical Study, pages 157–170. Springer-Verlag, Berlin, Heidelberg, 2014.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and d. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, pages 211–221. IEEE Press, 2012.

[11] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, (5):510–518, 1981.

[12] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 371–372. ACM, 2010.

[13] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[14] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[15] M. Lorenz and J. Kidd. *Object-oriented software metrics*, volume 131. Prentice Hall Englewood Cliffs, 1994.

[16] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[17] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 7. ACM, 2008.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[19] B. A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[20] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33. ACM, 2018.

[21] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 91–100. IEEE, 2014.

[22] J. Ratzinger. *sPACE: Software Project Assessment in the Course of Evolution*. PhD thesis, 2007.

[23] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[24] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016.

[25] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, and M. Cornlio. Saferefactor-tool for checking refactoring safety. 01 2009.

[26] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, 2013.

[27] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, pages 10–10. IEEE, 2007.

[28] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. 2018.

[29] D. Zhang, B. Li, Z. Li, and P. Liang. A preliminary investigation of self-admitted refactorings in open source software. 07 2018.