On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns

Vinícius Soares,
Anderson Oliveira,
Juliana Alves Pereira
PUC-Rio, Rio de Janeiro, Brazil
[vsoares,aoliveira,juliana]@inf.puc-rio.br

Ana Carla Bibano,
Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
[abibiano,afgarcia]@inf.puc-rio.br

Paulo Roberto Farah[1,2]
Silvia Regina Vergilio[1]
[1]DInf-UFPR, Paraná, Brazil
[2]UDESC, Santa Catarina, Brazil
[prfarah,silvia]@inf.ufpr.br

Marcelo Schots
Universidade do Estado do Rio de Janeiro,
Rio de Janeiro, Brazil
schots@ime.uerj.br

Caio Silva,
Daniel Coutinho
PUC-Rio, Rio de Janeiro, Brazil
[csilva,dcoutinho]@inf.puc-rio.br

Daniel Oliveira,
Anderson Uchôa
PUC-Rio, Rio de Janeiro, Brazil
[doliveira,auchoa]@inf.puc-rio.br

ABSTRACT

Developers need to consistently improve the internal structural quality of a program to address its maintainability and possibly other non-functional concerns. Refactoring is the main practice to improve code quality. Typical refactoring factors, such as their *complexity* and *explicitness* (i.e., their self-affirmation), may influence its effectiveness in improving key internal code attributes, such as enhancing cohesion or reducing its coupling, complexity and size. However, we still lack an understanding of whether such concerns and factors play a role on improving the code structural quality. Thus, this paper investigates the relationship between complexity, explicitness and effectiveness of refactorings and non-functional concerns in four projects. We study four non-functional concerns, namely maintainability, security, performance, and robustness. Our findings reveal that complex refactorings indeed have an impactful effect on the code structure, either improving or reducing the code structural quality. We also found that both self-affirmed refactorings and non-functional concerns are usually accompanied by complex refactorings, but tend to have a negative effect on code structural quality. Our findings can: (i) help researchers to improve the design of empirical studies and refactoring-related tools, and (ii) warn practitioners on which circumstances their refactorings may cause a negative impact on code quality.

CCS CONCEPTS

• **Software and its engineering → Extra-functional properties**; *Maintaining software*; *Software development techniques.*

KEYWORDS

refactoring, internal quality attributes, self-affirmed refactorings, non-functional concerns

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422439

ACM Reference Format:
Vinícius Soares,, Anderson Oliveira,, Juliana Alves Pereira, Ana Carla Bibano,, Alessandro Garcia, Paulo Roberto Farah[1,2], Silvia Regina Vergilio[1], Marcelo Schots, Caio Silva,, Daniel Coutinho, Daniel Oliveira,, and Anderson Uchôa. 2020. On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422439

1 INTRODUCTION

The refactoring activity consists of applying one or more refactoring transformations types, focusing on improving code quality as a means to achieve better maintainability [16]. Along software maintenance, developers perform improvements to non-functional requirements (NFRs) while also applying well-known code refactorings [21]. Even though developers often apply refactorings in practice, concerns about refactoring and NFRs are rarely explicitly mentioned along a change. Thus, one might wonder whether refactoring is more effective when an explicit manifestation of such concerns occurs. A refactoring is considered effective when it successfully improves internal code quality attributes [8, 16, 19, 26], such as enhancing cohesion or reducing coupling, complexity and size. Internal quality attributes are not only the academy-standard metrics for detecting problematic code [2]. Empirical studies (see Section 2) also show a relationship between these quality attributes and various NFRs beyond maintainability, such as security, performance and robustness. However, there is no knowledge if well-known refactorings [16] are more effective when developers perform changes with such NFR concerns in mind.

In addition, even though refactorings were proposed as singular transformations, developers often apply they in conjunction trough the so-called *batch* or *composite* refactorings. The joint application of various transformation types – i.e., more complex refactorings – might increase the likelihood of effectively improving code quality. Composite refactorings comprise about half of the refactorings applied in software projects [6, 24]. Thus, their research has been growing in popularity in the last few years [6]. Along with this, recent studies also described the many kinds of composite refactoring patterns used in practice [34]. They range from a simple, repeated

usage of one or two different types of refactorings, to complex processes spanning over 5 different types of refactorings [7].

One could hypothesize that refactorings are expected to be more effective in improving code structural quality based on (i) the combination of multiple transformation types ("complex refactorings") in order to solve major structural problems in the code [6, 24, 34]; (ii) explicit mentions of concerns with refactoring (refactoring explicitness) – which is also popularly known as self-affirmed refactorings (SARs) [1, 28]; and (iii) explicit mentions of concerns associated with NFRs (these concerns are referred to in this work as NFCs).

Studies show that refactorings are not always effective in terms of improving structural quality attributes [6, 7, 34]. However, it is still not known whether the expectations above have any relationship with the effectiveness of refactoring. Existing studies only focus on analyzing if a refactoring is effective or not; they do not investigate to what extent their complexity, as well as their associated concerns, relates to its effectiveness. For example, a complex SAR with an explicit concern with maintainability (and other NFRs) might have a positive impact on the software's structural quality.

In this context, this study intends to conduct a preliminary investigation of whether and how the variation of refactoring complexity and its explicit concerns correlates to the improvement of internal quality attributes. To this end, we analyzed a total of 2,588 refactorings, obtaining information on both their effects on internal quality attributes and the amount of unique refactoring types they are composed of. We also developed and evaluated two keyword-based classifiers, one for SAR, and one for NFC detection in developer discussions – with the SAR detector reaching an F1-Score of over 80%. Through this analysis, we achieved the following findings:

Refactoring Complexity. We were able to determine that the complexity of a refactoring is impactful on their effects. As refactoring complexity increases, so does the chance of positive impacts; however, so does the risk of having a negative impact on the code. This reinforces the notion that refactoring complexity should be considered in studies and refactoring techniques. Thus, with proper guidance, a more effective means of refactoring can yield better results than what is currently being considered in the academy.

Refactoring Explicitness. An explicit concern with refactoring, surprisingly, more frequently affects their effectiveness negatively. Though many of the explicit refactorings were more complex, the developers did not always select refactoring type compositions that have shown themselves to be adequate to the improvement of structural quality. Less complex refactorings, composed of at most three types, had more positive effects – even if they were less effective overall. These results suggest that developers might need to be more concerned with the process they follow when implementing complex refactorings, and not only the act of refactoring itself, as the improvement of code structure is not being achieved. These results could also be explained by the lack in tool support for complex refactorings, similar to what was found in other works [21, 35, 37].

Non-Functional Concerns. An explicit concern with NFRs may actually hinder the application of refactorings. However, when concerns with maintainability and robustness were present, refactorings generally had a more positive effect, while performance and security brought positive impacts on only one specific attribute each. This may suggest a need for more optimization-based recommenders [25] to aid developers in making changes that positively affect necessary metrics without causing detriment to the others.

2 RELATED WORK

This section classifies related works into three categories: *effectiveness*, *explicitness*, and *non-functional concerns*.

Effectiveness. Alshayeb [3] performed an empirical analysis of three Java systems, in order to evaluate claims that refactoring improves software quality. They evaluated possible correlations of the internal quality attributes of cohesion, coupling, (code) complexity, inheritance and size with the external quality attributes such as maintainability and testability. They concluded that refactorings rarely had positive effects on these attributes, being mostly neutral. Similarly, on the same context of internal quality attributes, Bavota et al. [5] studied whether internal quality attributes or code smells relate with refactoring needs, based on 11 attributes (including size, coupling and cohesion) and 10 types of smells. Results showed that the analyzed refactorings focused on changing components whose quality metrics did not indicate problems in code.

Once again, regarding internal quality attributes, Chávez et al. [12] investigated how root-canal and floss refactorings relate with internal quality attributes, including cohesion, coupling, (code) complexity and inheritance. They found that over 94% of applied refactorings have negative impacts on at least one internal quality attribute. Moreover, most refactorings improve their quality attributes, while others keep them unaffected.

On the context of complex refactorings, Bibiano et al. [7] performed a study on 5 different open-source projects, in order to analyze the impact of incomplete composite refactorings in their quality attributes. They found that most incomplete composites have a neutral effect on internal quality – neither increasing nor decreasing code quality. Also on the context of composite refactorings, Fernandes et al. [15] evaluated how composites compare to single refactorings in improving cohesion, (code) complexity, coupling, inheritance and size of affected elements. Among the analyzed refactorings, 65% improved attributes associated with their refactoring types, while 35% kept them unaffected. Thus, while the aforementioned works studied potential correlations between refactoring effectivness and other factors, they did not consider refactoring complexity and the presence of NFCs as potential factors.

Explicitness. The presence of SARs in software development has been only recently explored – with the term "self-affirmed refactoring" being coined in the last few years. Ratzinger [28] proposed a phrase-based approach to detecting developer discussions related to refactorings. In the explored context, the goal was the prediction of potential refactorings, which they achieved with a high accuracy. Likewise, for SAR-related research, AlOmar et al. [2] performed an empirical study on SARs in order to identify if academia-standard design metrics reflect what developers consider as quality. Results indicate that, for cohesion, coupling, (code) complexity, and inheritance, the academia-standard metrics do reflect developers' definition of quality. However, for encapsulation, abstraction, and design size, there is a mismatch on how metrics were proposed and how they are used in practice. While both of the aforementioned works analyzed the presence of SARs, they did not analyze a potential

correlation with the complexity and effectiveness of refactorings performed in the code.

Non-Functional Requirements The relationship between internal quality attributes and NFRs is addressed by a number of works. Thus, we focus our analysis on works addressing the NFRs that compose the NFCs analyzed in this work.

Regarding performance, Siegmund et al. [31] analyzed refactorings' effects on the performance of software product lines. Refactorings such as *Inline Method* and *Inline Class* can reduce the execution time on method calls, thus improving performance. Also, Götz and Pukall [18] showed that removing code delegation and indirection can improve software performance by around 50%. Demeyer [14] reported performance improvements after replacing conditional statements with call methods through polymorphism. Some works also analyzed the relation between size, defined in terms of code statements, and performance [4, 27]. Smith and Williams [32] discussed performance anti-patterns based on coupling, cohesion, and (code) complexity of the inheritance hierarchy, concluding that God Classes are detrimental to software performance.

In the context of robustness, it is common to use exception flow information [9, 10]. It aims to improve code reliability by providing constructs for sectioning code into exception scopes (e.g. try blocks) and exception handlers (e.g. catch blocks). Jakobus et al. [20] evaluated the robustness of 50 projects by using the internal quality attributes of size and (code) complexity. Their findings suggest that exception handlers are usually simplistic, and that developers often pay little attention to exception scoping and behavior handling.

We also found works relating internal quality attributes to security. Chowdhury et al. [13] evaluated how internal quality metrics of coupling, cohesion, and (code) complexity can indicate security risks in software. They concluded that size metrics can indicate structures that could be exploited to cause a denial of service attacks, while coupling can impact on how damage may propagate to other components of the software. Yet, their results showed that these metrics are not sufficient to indicate specific vulnerability types. Other studies[23, 30] also support these findings.

The identification of NFCs in text and developer discussions has also been explored. Lu and Liang [22] proposed an automatic classification of user reviews into concerns with four NFR types: reliability, usability, portability and performance. They evaluated the combinations of the classification techniques and machine learning algorithms with user reviews collected from two popular mobile apps from different platforms and domains. Results show that a combination of algorithms achieves an F-measure of 71.8%. Casamayor et al. [11] applied a semi-supervised learning approach to identify NFCs in textual specifications, using a collection of requirements-related documents from 15 different software development projects, consisting of 370 mentions to NFRs and 255 functional ones.

These works differ from our work because, while they attempted to collect NFCs from project development history, they did not explicitly investigate the association of refactoring complexity, effectiveness, explicitness, and NFCs.

3 METHODOLOGY

In this section, we describe the methodology adopted in our study. The main goal of this work is to investigate to what extent the refactoring complexity, as well as their associated concerns (explicitness and NFCs), relates to its effectiveness, that is, to the improvement of internal quality attributes in the software.

3.1 Research Questions

Our analysis is guided by the following three research questions:

- **RQ1. Is the complexity of refactorings related to their effectiveness?** RQ1 is motivated by a search of a potential correlation between the complexity and effectiveness of refactorings. Thus, with this RQ, if this correlation exists, we aim at understanding its nature. We define complexity as *the number of different refactoring types that compose the applied refactoring*. Similarly, we define effectiveness as *the impact of the refactoring in improving internal quality attributes by improving their associated metrics*. We fully explain the reasoning for these definitions in Section 3.4.
- **RQ2. Are refactorings' complexity and effectiveness related to their explicitness?** Since the complexity of refactorings may not be the only factor that affects their effectiveness, RQ2 is built on top of the idea that this other factor may be the developers' explicit concern about refactoring. This explicitness is defined as *the presence of a SAR in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied*. If this explicitness is related to the refactoring complexity and effectiveness, this RQ aims at understanding how it takes place.
- **RQ3. Do NFCs relate to refactoring effectiveness?** Finally, RQ3 aims at investigating if there are other concerns that affect the effectiveness of code refactoring. Thus, we define NFCs as *the presence of one of four analyzed NFRs in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied.*

The choices and artifacts analyzed to answer the research questions are listed as follows:

Selection of Internal Quality Attributes. We chose to individually analyze each of the four internal quality attributes of cohesion, complexity, coupling and size for two reasons: (i) their connection to the NFRs chosen for this work (*e.g.*, size and complexity correlates to performance), and (ii) due to their uses in other works [7].

Selection of Non-Functional Requirements. *Maintainability* was chosen due to its potential likeliness of influence in refactoring, *i.e.*, it can be considered as an usual concern, as according to previous works. The other three NFRs were selected because they strongly relate with refactorings. The increase of *security* can be linked to refactoring strategies that redesign application structures; preventing intruders from accessing sensitive code commands. Improvements in *robustness* can be reached by the reorganization of modules for integrating patterns geared at error handling (e.g, Chain of Responsibility [17]). Finally, *performance* can be increased through the detection, and subsequent refactorings, of code redundancies or a suboptimal distribution of code entities.

Figure 1 summarizes the methodology adopted to answer the research questions. First, we selected four projects, based on the criteria described in Section 3.2. Second, we collected data regarding internal quality attributes, refactorings and developer discussions, which is described further in Section 3.3. Third, we performed an analysis of the collected data to obtain composite refactorings (see Section 3.4) – which combine two or more refactoring types. In addition, we collected data about refactoring effectiveness, as well as the presence of SARs and NFCs. Finally, we used this data to answer each of the aforementioned research questions.

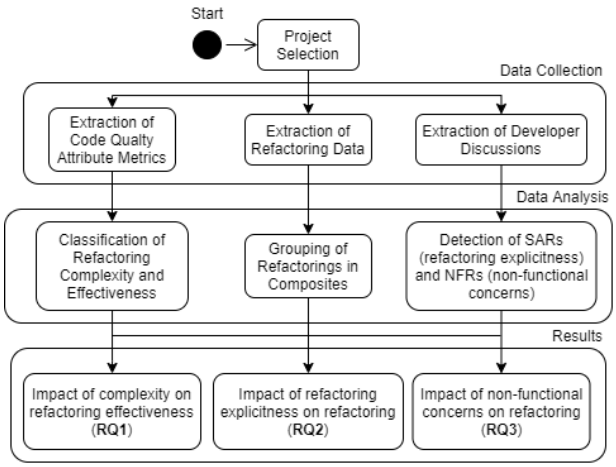

Figure 1: Adopted methodology.

3.2 Project Selection

Due to our focus on understanding the impact of complex refactorings in real-world projects, we selected 4 projects – Couchbase Java Client, Dubbo, OKHttp and JGit –, using the following criteria:

Java Open Source Projects. The project must be open-source, and developed primarily in Java. This increases this work's replicability. Alongside this, Java has the most support from outside tools, especially for internal quality analysis – which are required for this work; **Number of Contributors and Activity.** The project must still be actively worked on at the time of analysis, and must have a considerable number of contributors. Active projects are more likely to accurately represent the state of the industry. Similarly, the large amount of active contributors in these projects allow us to have a larger selection of discussions; **Project Age.** The project must have been in development for at least 5 years. Architecture degradation over time is more clearly seen in older projects, which would then require developers to more frequently apply refactorings.

Variety of Refactoring Types. The project must have a large variety of different refactoring types (*i.e.*, one of the refactorings as defined by Fowler [16]) being used along its maintenance. This is important due to our focus that is on refactoring complexity – which is defined as the amount of different refactoring types used in a composite or single refactoring; **Presence of Composite Refactorings.** The presence of composite refactorings in the project must have been proven in other works. Once again, due to our focus being on the analysis of complex refactorings, the presence of composite refactorings in the analyzed projects is important. Thus, we decided to choose projects in which the presence of composite refactorings was already proven by other works, such as [7].

3.3 Data Collection

Thus, in order to attain the goals described previously, we extracted data from three main points of view: (i) *Internal Quality Data*, which is necessary for all **RQ**s; (ii) *Refactoring Data*, which is also necessary for all **RQ**s, and; (iii) *Developer Discussions*, for **RQ2** and **RQ3**. Their collection would enable their correlation, thus allowing an understanding of refactoring complexity's effect on internal quality attributes. It also allows an understanding of a correlation between the presence of SARs and NFCs in changes that contain refactorings, and the complexity and effectiveness of these refactorings. The collection of this data was done as follows:

Internal Quality Data. In order to obtain data related to internal quality attributes, we must first collect the metrics that compose such attributes. To this end, we used a tool called Understand [29]. Understand is a static code analysis tool, which collects the internal quality metrics from each element, in each commit, of each project. This then allows us to view a continuous trend in the change of each metric in the project's history. In turn, this enables a detection of how changes improved – or worsened – the state of each attribute. However, Understand has a very large amount of metrics related to the size and complexity of the code, and lacks metrics for code cohesion. Thus, in order to balance the amount of metrics in each attribute, we selected a set of 15 metrics, which are listed in the work's companion website [33].

Refactoring Data. For the data set related to the second point of view, we focused on which kinds of refactorings were used during the project's development, as well as how they form composites. In order to detect and classify refactorings based on the types proposed by Fowler [16], we used RefMiner [36], a tool that collects refactoring information from the history of Java projects. RefMiner has a high reported precision of 98%, and a recall of 87% [36], which makes it a reliable tool for refactoring detection.

Developer Discussions. Finally, for the third point of view, we focused on attempting to extract the messages and discussions written by developers when changing the code. Thus, we extracted the following items from the projects' repositories, by using the GitHub API: (i) Commit Messages; (ii) Related Issues and Pull Requests; and (iii) Developer Comment Discussions. While we were able to extract commit messages for all 4 projects, 2 of them (Couchbase Java Client and JGit) did not have Issue and Pull Request information on their GitHub repositories, since they utilize an external issue/review tracker. However, this distinction in information availability may prove useful in understanding how much SAR and NFC detection is capable of detecting potential candidates with limited information.

3.4 Data Analysis

Once the aforementioned data was collected, we started the process of data analysis. While the main steps and results are described in this paper, additional information about it can be found in the work's companion website[33]. The analysis consisted of combining internal quality attribute and refactoring complexity data with the presence of SARs and NFCs. Thus, this allowed us to answer the proposed **RQ**s (Section 3.1). This analysis was performed as follows:

Refactoring Complexity and Effectiveness. Once we had access to both refactoring-related and internal quality attribute data, we were able to determine what would consist as "refactoring complexity", and how this would affect their effectiveness. Originally, we intended to determine complexity as a combination of how many refactoring types were used in a composite/single refactoring, as well as how many code elements were affected. This metric for affected elements was defined as the amount of unique classes and methods related to the refactoring in question. However, we decided against using this metric, as it did not have any relation with refactoring effectiveness in our analysis. Another potential complexity metric for refactoring exists – the amount of single refactorings that are part of the same composite, regardless of type. Nonetheless, we decided against using it, as it was already seen to not affect refactoring effectiveness in other works [6].

Thus, we defined the metric for refactoring types as the number of unique detected refactorings that composed either a composite or single refactoring. In total, we were able to detect up to 52 unique refactoring types, at method and class level – such as *Extract Class*, *Move Method* and *Split Attribute* –, through the use of RefMiner. The full list of refactoring types is in the work's companion website [33].

To determine refactoring effectiveness, we analyzed each of the four internal quality attributes (complexity, cohesion, coupling and size) individually, analyzing how a change in other parameters (refactoring complexity, presence of SARs and NFCs) affected each attribute. In order to classify a change as *positive*, *negative* or *neutral*, we used the following criteria: (i) if there was no positive (reduction) nor negative (increase) change in any metric related to the attribute, the change was *neutral*; (ii) if at least one of the metrics related to the attribute had its value changed, and most metrics changed positively, the change was classified as *positive*; (iii) in any other case, it was classified as *negative*. Thus, by combining this effectiveness data with complexity data, we could better understand if they relate with each other, and if so, how they relate. Thereby, we were able to answer **RQ1**.

Composite Refactoring. Due to analyzing refactoring complexity – we needed to group the previously obtained refactorings into composites. To do so, we first used the *range-based* heuristic, as defined by Sousa et. al. [34]. This heuristic groups refactorings that affect similar elements in different points of time as a composite, which would allow us to analyze changes that spanned multiple commits. Afterwards, we grouped the remaining refactorings into composites through a *commit-based* heuristic, also proposed by Sousa et. al. [34]. This heuristic groups refactorings that were applied in the same commit as a composite refactoring. Then, the remaining refactorings were classified as single refactorings. Thus, we were able to group the highest possible amount of interrelated refactorings into composites, in order to better detect the complexity of the refactorings used by the developers when changing the code. By doing so, we can combine this data with the internal quality attribute metrics in order to answer **RQ1**.

Presence of Self-Affirmed Refactorings. By using the previously collected developer discussions, we were able to automatically detect and classify which refactorings had a correlated discussion in which a SAR was present. This was done through keyword-matching, by using a set of 11 keywords and 8 phrases, which are listed in the work's companion website [33]. The original set of keywords was based on Ratzinger's work [28], though we changed the set in order to improve its accuracy for the data set we used in this work. Thus, by combining this data with the previously collected complexity and effectiveness data, we answered **RQ2**.

Non-Functional Concerns. By using the previously collected developer discussions, we were able to automatically detect and classify them if they are NFCs or not. This was also done automatically through keyword-matching, by using a set of 69 keywords, which are listed in this work's website [33]. We decided on the keyword set based on previous manual analyses of the projects' issue/PR messages. However, as we describe further in Section 4, this NFC classifier has a very low accuracy, but a relatively high recall.

Considering this, we decided to focus our analysis on a manually validated refactoring sample, designated by the classifier. Thus, we had a final analyzed group of 196 refactorings. From this group, 92 refactoring changes did not mention NFRs, while the other 104 did. In addition, 33 refactorings were related to maintainability, 20 to security, 36 to performance, and 40 to robustness – with potential intersections between these classifications. As such, the combination of this manually-validated data with all the aforementioned data sets allows us to answer **RQ3**.

4 VALIDATION

With the goal of determining the reliability of the automatic detection of SARs and NFCs, we performed manual validations for each, determining their precision and recall.

4.1 Self-Affirmed Refactoring Validation

For the manual validation of SARs, we selected a set of 124 different commits split into 63 commits from OKHttp, and 61 commits from Couchbase Java Client. The set was also divided equally between three different groups: (i) commits classified as SAR, and also classified as refactorings by RefMiner; (ii) commits classified as SAR, but not classified as refactorings by RefMiner, and; (iii) commits that were not classified as SARs.

We decided to validate commits from these two projects for the following reasons: (i) both have a high frequency of SARs in relation to other projects – enabling a more accurate check of which keywords are problematic and which are missing; and (ii) While OKHttp has available information for commits, issues, pull requests and comments, Couchbase Java Client only has commit messages available. Thus, we can test if an actual difference exists in the accuracy when less information is available. Likewise, the three groups were decided for the following reasons: (i) ensuring that all keywords in the set were verified. To do so, we selected SARs from the keyword sets with less occurrences, and only later did we select those with more common keywords; (ii) being able to detect if an actual refactoring was present when a SAR was falsely detected.

We then performed a manual validation with 3 participants – all 3 being authors of this work, and knowledgeable in the context of refactoring. 2 authors classified a set of 42 random commits, while one last author classified a set of 40. These classified sets were also equally balanced between the three sets described previously. In this validation, the participants were asked to identify the following: (i) if there was any explicit mention of a term or phrase that would

lead to a direct association to a refactoring; (ii) which phrase led them to understand identify the SAR, and; (iii) which keywords were related to it. The full results of this validation are available in this work's companion website [33].

Through the completed validation, we can quantify the precision and recall of the SAR detector. From this data set, it had a precision of 81.2%, and a recall of 91.5% – leading to an F1-Score of 86%. Individually, the precision was slightly lower for Couchbase Java Client in relation to OKHttp. However, the classifier had over 80% recall in both projects. This can mean that the unavailability of other kinds of discussions in two of the four projects analyzed in this work might not have much impact in the final results.

4.2 Non-Functional Concern Validation

Similarly to the SARs validation, we also made efforts in attempting to manually validate the NFC classifier – which was needed, due to the complexity involved in identifying NFCs on textual data. First, we performed a validation of 1200 issues from OKHttp. The data was split into those that were detected as NFC and those that were not. This first validation was made with 6 participants, all of them computer science master/doctorate students knowledgeable in the context of NFRs. They were asked to classify the issues as one (or more) of the NFCs used in this study (maintainability, robustness, security and performance). They were also asked to determine which phrase led them to their decision, and which keywords in that phrase could be used to identify and classify that NFC.

Once again, we chose OKHttp due to the high frequency of NFR-related discussions in that project, as well as the presence of issues and pull requests as potential discussions to be analyzed. However, we found that the NFC classifier precision was below-average (50.43%), even if the recall was good (82.66%) for this data set. As such, we thought that, by allowing the classifier to focus on commits in which changes actually occurred, and not only issues and pull requests (which could be closed without any action in the code itself), we could more accurately classify in terms of their related concerns.

Thus, we then specialized our validation for a new set of 553 commits in which refactorings occurred – in order to perform the analysis for finding a potential trend between the presence of NFCs and refactoring effectiveness. This data set was split between three projects: OKHttp, Couchbase Java Client and JGit. Thus, we could balance the set between projects in which we have full information available, as well as projects in which we have only commit information. We also attempted to balance the data set between mentions to all four detected NFRs.

This validation was performed by 5 participants – once again, with all of them being computer science master/doctorate students knowledgeable in the context of refactoring. The validation process was very similar to the previous one, though now the participants were also asked to describe how directly the NFR was mentioned (if it was). In this case, they should identify if the NFR was directly mentioned in the discussion, or if it could be indirectly derived from the text under analysis.

Through this second validation, we can quantify the precision and recall of the NFC classifier – leading to a precision of 53.90% and a recall of 72.13%, correlating to an F1-score of 61.70%. The low precision might make the use of the automatically classified data not a good approach for a quantitative analysis. However, the good recall makes using it to collect a sample of NFC discussions for validation – as discussed in Section 3.4 – a valid approach.

5 RESULTS AND DISCUSSIONS

In this section, we present the results that answer the **RQ**s introduced in Section 3.1. We begin by analyzing the correlation between refactoring complexity and effectiveness (**RQ1**), then correlating our findings to the presence of SARs (**RQ2**) and, finally, correlating all findings to the presence of NFCs (**RQ3**). The numerical results for each of the analyses described in this paper are available on the project's website [33].

5.1 Refactoring Complexity vs. Effectiveness

At first, we attempted to understand how much the increase in refactoring complexity affects each internal quality attribute of the refactored code. To this end, we grouped refactorings into 5 categories, based on the number of different refactoring types they were composed of. Categories 1 to 4 contained refactorings with 1 to 4 refactoring types, respectively, while category 5+ contained those with 5 or more refactoring types, since these were rare occurrences and, by combining them, we have a more balanced data set.

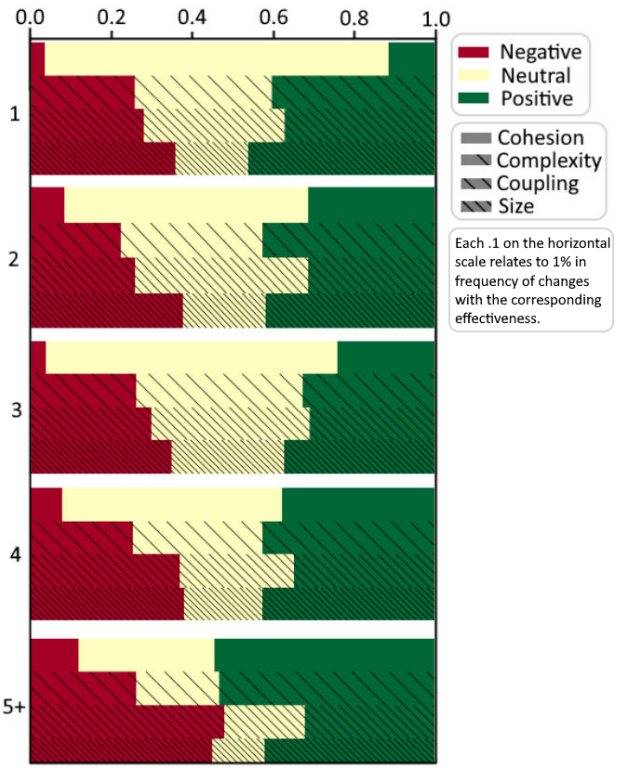

Figure 2: Distribution (decimal percentage) of effects based on the refactoring complexity.

Figure 2 shows that, for the more complex refactorings – containing 4 or 5+ refactoring types –, the proportion of refactorings

that had neutral effects on the code decreased in all 4 attributes. In average, the difference between 1 to 5+ was 21.2%, with cohesion having a drastic reduction of 51.4%, and high reductions in coupling and complexity. Conversely, the proportion of positive and negative changes also increased. We can also notice that, proportionately, the frequency of negative changes increased more than those of the positive changes. In some cases (namely coupling and size), the number of negative changes even overcame the number of positive changes. We observed a significant difference in the distribution of effectiveness between refactoring complexity levels for all quality attributes. The difference is with 95% confidence ($\alpha = 0.05$) using Kruskal-Wallis Chi-Squared test and Dunn's post-hoc test with Bonferroni adjustment, except for cohesion (p-value > 0.6). These increases in the magnitude of effects for more complex refactorings follow what would be expected – since, when adopting more types of refactorings, the possibility of making significant improvements (or deteriorations) in code quality attributes is higher. Thus, complex refactorings could be a risky, but potentially fruitful, endeavor.

Conversely, this increased complexity could also lead to more mistakes during its execution, and thus potentially cause an increase in negative effects. This increase in negative effects we experienced, thus, may be due to the lack of tool support for the application of complex refactorings spanning multiple refactoring types.

> **Finding 1**: **(RQ1)** The more complex the refactorings are, the higher their impact on structural code quality. This impact can be both positive and negative.

Implications. Our results are interesting, since previous studies that analyzed other ways of defining refactoring complexity – such as the number of refactoring instances in a composite [6], or the number of commits in a refactoring [6, 34] – found no relation between these complexity indicators and internal quality attributes. In contrast, our results indicate that the diversity of types in a refactoring tend to impact internal quality attributes. As such, future studies may consider this indicator when evaluating the impact of refactoring complexity in internal and external quality attributes. From a practical point of view, developers can benefit from our findings of how the addition of more refactoring types to a composite affects the code: the increase in the risk of causing negative changes, yet the potential for more frequent positive changes.

5.2 SARs vs. Complexity and Effectiveness

By using the information of which refactorings were self-affirmed or not, we are able to combine them with our previous results to uncover more findings. To this end, we analyzed the correlation between refactoring complexity and its self-affirmation, through SARs. Thus, Figure 3 presents the frequency of self-affirmed (Has SAR) and non self-affirmed (No SAR) refactorings composed of 1, 2, 3, 4, or 5+ refactorings. It can be seen that, while comparing the SAR and No SAR sets, respectively, the proportion of single refactorings severely decreased (57% to 34%), while the proportion of complex changes with 5 or more refactorings severely increased (7% to 28%). The results are statistically significant, using Wilcoxon Rank-Sum test at 95% confidence level ($\alpha = 0.05$).

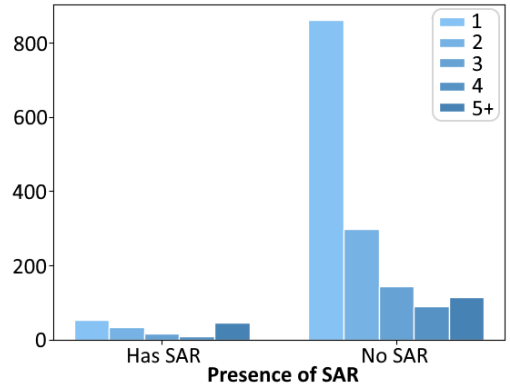

Figure 3: The frequency of self-affirmed and non self-affirmed refactorings composed of 1, 2, 3, 4, or 5+ refactorings.

Thus, our findings reveal that, when the primary focus of developers is to perform a refactoring – which could be indicated by a self-affirmation of their refactoring [2] – they tend to perform more complex, and thus, more impactful, refactorings.

> **Finding 2**: **(RQ2)** Developers tend to perform more complex refactorings when they manifest an explicit concern with refactoring.

Figure 4 presents the effectiveness of self-affirmed (Has SAR) and non self-affirmed (No SAR) refactorings. As discussed, we define effectiveness as the improvement of cohesion, complexity, coupling and size. As in Section 5.1, the frequency of neutral changes also decreased, in average, 7.4%. The results for effectiveness between SARs vs non-SARs are statistically significant using Wilcoxon rank sum test at 90% confidence level, except for cohesion (80% level).

Moreover, we also observed that the proportion of negative changes for SARs increased much more expressively (more than ten times) than the proportion of positive changes. Thus, our results indicate that having refactorings as the developers' primary focus may actually hinder their effectiveness – potentially due to this increase in complexity causing more mistakes in the refactoring process. One possible reason for these results may be due to the lack of tool support for applying complex refactorings, as described in other works [21, 35, 37]. Therefore, we can expect that developers are performing refactorings mostly manually. Thus, we can summarize the aforementioned results as the following finding:

> **Finding 3**: **(RQ2)** Self-affirmed refactorings tend to have a more negative effect on the code than their counterparts.

Implications. Our results indicate that, when developers explicit their concern in the refactoring process, they tend to do more complex refactorings – though they also tend to perform worse. This may imply that developers have a higher concern with the refactoring process when they apply complex refactorings. Thus,

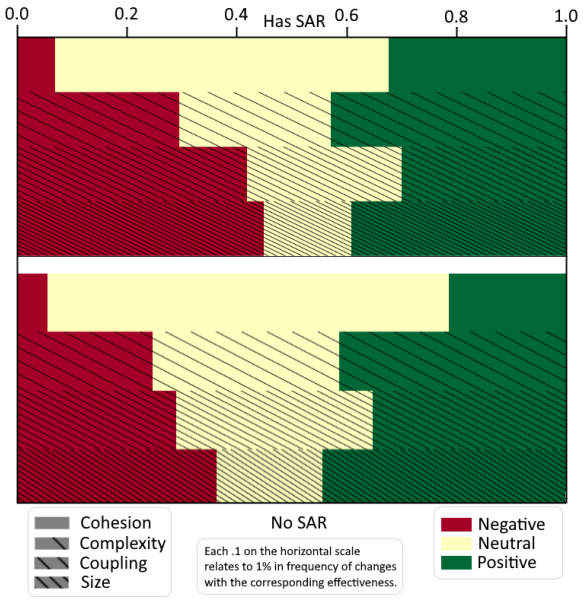

Figure 4: The negative, neutral and positive effect of self-affirmed and non self-affirmed refactorings.

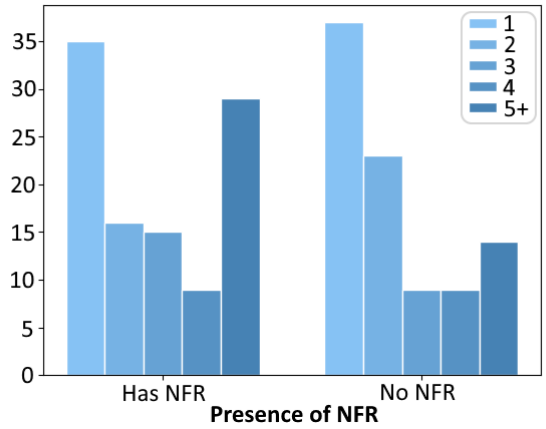

Figure 5: The frequency of refactorings composed of 1, 2, 3, 4, or 5 or more refactorings grouped by the presence of mentions to NFRs.

they tend to explicit their refactorings concern on the commit messages. Consequently, it is surprising that they also yield more negative results. Knowing this, developers should be more aware, beyond just the concern they demonstrated with refactorings, of the process they follow when implementing refactorings. This is important, as the main basic objective of refactoring, improving the structure of the refactored code, is not being reached.

Interestingly, we also found that over 30% of the validated sample was detected as an SAR by both the manual validation as well as the automatic detector – but was not detected as a refactoring by RefMiner. Alongside this, over 50% of the total commit messages (spanning all projects) were detected as SARs and were not detected by RefMiner. Thus, this might mean that developers frequently apply refactorings that differ from those defined by Fowler [16], and are thus not detected by RefMiner. We also observed that developers often customize even single refactorings, similar to was reported by Tenorio et al.[35] – which would impact their detection by RefMiner, as they do not follow the steps defined by Fowler [16].

5.3 NFCs vs. Complexity and Effectiveness

In this section, we perform an analysis of how NFCs related to each of the four NFRs (maintainability, security, performance and robustness), can affect both the complexity and the effectiveness of their related refactorings. As described in 3.4, we only used a set of 196 refactorings for this analysis, split into a set of 92 which do not contain mentions to NFRs, and a set of 104 that do. Thus, Figure 5 presents the frequency of refactorings composed of 1, 2, 3, 4, or 5 or more refactoring types – divided into a set of changes in which there was an explicit concern with one of the four NFRs, and one in which there was not.

From this, we can observe that, very similarly to when refactorings are explicitly mentioned, complex refactorings are more frequent when the developers express an explicit concern with NFRs. We found no statistical difference (p-value > 0.6) between them. This is potentially a result of the the NFR detector's low accuracy, leading us to use a small validated data set – further motivating a search for more effective automated NFR detection. Thus, this can be summarized into the following finding:

> **Finding 4**: **(RQ3)** Developers usually perform more complex refactorings when they are explicitly attentive to NFRs.

Finally, we analyzed the impact of refactorings when coupled with NFCs. First, by analyzing the difference between NFR-related changes for each of the 4 NFRs, we saw that the increase in the negative effects of refactorings was significant – with no apparent increase in its positive effects. However, positive changes were still more frequent than negative changes. This correlates to the findings described in Section 5.2 – in which developer concerns reduce the neutral effects of refactorings, but also increase the possibility for negative effects. We also focused on the analysis of how mentions to each individual NFR impacts on the effectiveness of refactorings. Figure 6 displays the results of this analysis, by showing how each concern relates to affecting the internal quality attributes. The attributes were analyzed individually by grouping their related metrics.

As expected, refactorings related to maintainability affect all internal quality attributes rather equally, and with more positive than negative effects, in general. Refactorings related to robustness also have a similarly-distributed effect, though with less pronounced negative effects. This is interesting, as maintainability would be the NFR most related to refactoring itself, yet it tends to cause more negative changes, in general, when compared to robustness.

Conversely, refactorings related to either security or performance are very focused – tending to change a single attribute very positively, while negatively affecting all others. In the case of security, we found that changes were usually focused in improving cohesion between elements. Following the principle of information

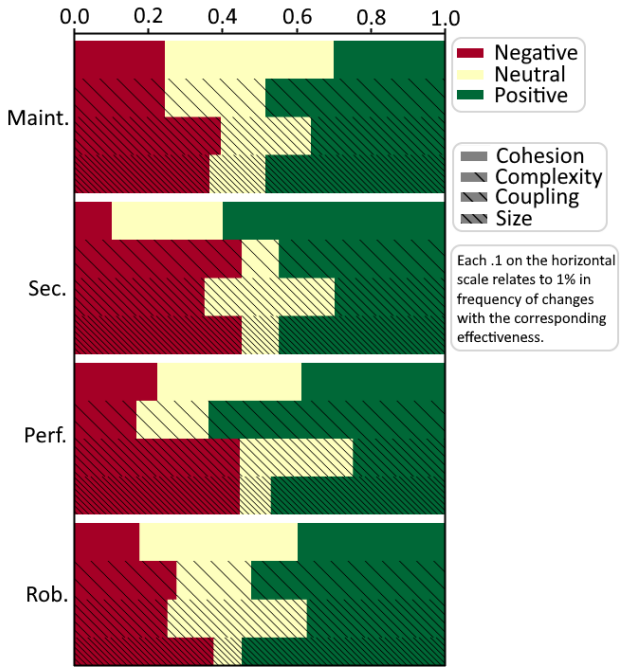

Figure 6: The negative, neutral and positive effects of refactorings when coupled with changes in NFRs.

hiding – it is expected for security-related changes to improve the cohesion of the code, by keeping information close to only the classes that use them. However, the fact that they often increase coupling in the code is surprising as, according to Chowdhury et al. [13], high coupling can be considered a security flaw, as it may cause a higher propagation of potential attacks.

In the case of performance, we found that changes were usually focused on reducing code complexity. Some works also correlate performance to code complexity-related changes, such as the addition of polymorphism, even with an increase in the number of sentences [14]. This also corroborates with our findings, as we detected that refactorings related to performance have a high frequency of negative effects in size-related metrics. We found statistical significance for internal quality attribute changes between the NFR types at 95% confidence level for coupling, and at 65% level for cohesion. But, we did not find statistical difference for code complexity and size. Thus, we can summarize these results in the following finding:

> **Finding 5**: **(RQ3)** Refactorings concerned with maintainability and robustness have a somewhat positive effect distributed through all metrics. Refactorings concerned with performance and security, however, have very focused effects – drastically improving one attribute with respect to the others.

Implications. We observed that when developers explicit their concerns with NFRs, they tend to perform more complex refactorings, with usually negative impact. However, by looking more closely at the impact of each concern, we can see that, usually, maintainability- and robustness-related concerns still have a mostly positive effect on the code. Security- and performance-related concerns, however, tend to cause mostly negative effects on the code, even if they do improve one particular attribute severely. As such, we can derive that these other concerns that developers may have can cause them to disregard the refactoring process itself, thus causing the increase in the frequency of negative changes.

We can also derive that specific concerns may even cause developers to focus entirely on one specific attribute, with negative impacts over most other attributes. Thus, when implementing refactorings, developers should once again be more aware of the followed process – since balancing the improvement of necessary attributes without negatively affecting others may change the situation found in these results. This also motivates potential search-based solutions, since there might be a lack of external support for suggesting potential changes that attempt to balance improvement for most affected attributes.

6 THREATS TO VALIDITY

Although we attempted to mitigate them to the best of our ability, this work contains some threats to its validity, as follows:

Generalizability. We selected 4 projects from different fields (database, network, distributed computing and git integration), and from different developers. However, the results we found might still not be generalizable to other contexts (e.g., closed-source projects).

Accuracy of Refactoring Detection. Though RefMiner has high reported accuracy in many different works, we did not directly evaluate its accuracy for our set of projects. However, other works have used RefMiner for a similar set of projects [7], and have reported good accuracy in its detection.

Accuracy of Self-Affirmed Refactoring Detection. Though we did manually validate a sample of the data set in order to identify the detector's accuracy, it still relies on a keyword-based classification to determine whether or not a refactoring is self-affirmed. Thus, the detector might not be generalizable to other projects, and its accuracy might change with new updates to the analyzed projects.

Accuracy of NFC Classification. The accuracy of the NFC classifier is still below-average, even when applied to a specialized data set, which makes it unreliable for analysis. However, we attempted to mitigate this threat by only using a manually validated data set from a sample of instances detected by the classifier.

7 FINAL REMARKS

This work attempted to understand the relationships between: (i) refactoring complexity; (ii) refactoring effectiveness; (iii) refactoring self-affirmation, and; (iv) the presence of NFCs during the refactoring process, in improving internal quality attributes. We performed a quantitative analysis of 2648 refactorings, from four different open-source projects. Our results demonstrate that developers tend to apply more complex refactorings when they are explicitly concerned with either the refactoring process, or NFRs (i.e., security, performance, robustness and maintainability). We also observed that complex refactorings are both more impactful in affecting the code quality, and much riskier than single refactorings. This is due

to the fact that, the more complex the refactorings are, the higher their positive and negative impact on structural code quality.

These findings, when combined with other studies, such as the one proposed by Tenorio et al. [35], which proposes that automated refactoring tools may not currently provide support for customized refactoring, may in fact present an even more entrenched problem. These available tools in commonly-used IDEs only support simple, standardized single refactorings – having little or no support for complex, customized composites. A possible explanation for the drastic increase in the negative impacts of complex refactorings might be, thus, that they had to be performed manually, with little aid from supporting tools. Thus, our work intends to motivate tool developers into improving support for more complex refactorings composed of multiple different types.

Our results might also drive researchers to understand why developers tend to make worse refactorings when they explicitly mention their concern with refactoring. Our findings can also motivate practitioners and researchers in analyzing how to perform complex, yet effective, refactorings. This can be helpful, as complex refactorings seem to be more effective at actually impacting the code than their simpler counterparts.

As future work, we plan on further analyzing how other factors can impact the refactoring process. We also plan on investigating specific patterns of complex refactorings, and how they relate to their effectiveness. Finally, we aim at extracting developers' goals during the refactoring process to assess whether additional non-refactoring goals affect how developers refactor their code.

ACKNOWLEDGMENTS

This work was partially funded by CNPq (434969/2018-4, 141276/2020-7, 312149/2016-6, 141285/2019-2, 140185/2020-8, 104254/2019-0, 141054/2019-0, 131020/2019-6), CAPES (88887.473590/2020-00), CAPES/Procad (175956), CAPES/Proex (88887.373933/2019-00, 373892/2019-00), and FAPERJ (200773/2019, 010002285/2019).

REFERENCES

[1] E. AlOmar, M. W. Mkaouer, and A. Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *3rd IWoR*. IEEE, 51–58.
[2] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini. 2019. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. *arXiv preprint arXiv:1907.04797* (2019).
[3] M. Alshayeb. 2009. Empirical investigation of refactoring effect on software quality. *Information and Software Technology* 51, 9 (2009), 1319–1326.
[4] G. An, A. Blot, J. Petke, and S. Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1100–1104.
[5] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1 – 14.
[6] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim. 2019. A quantitative study on characteristics and effect of batch refactoring on code smells. In *ESEM*. IEEE, 1–11.
[7] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca, M. Ribeiro, C. Barbosa, and D. Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th ICPC*.
[8] F. Bourquin and R. K. Keller. 2007. High-impact refactoring based on architecture violations. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 149–158.
[9] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, and I. Garcia. 2014. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In *IEEE ICSME*.
[10] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E. A. Barbosa, and A. Garcia. 2014. Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs. In *36th ICSE*. ACM, 584–595.
[11] A. Casamayor, D. Godoy, and M. Campo. 2010. Identification of non-functional requirements in textual specifications: A semi-supervised learning approach. *Information and Software Technology* 52, 4 (2010), 436 – 445.
[12] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes? A Multi-Project Study. In *31st SBES*. ACM, 74–83.
[13] I. Chowdhury, B. Chan, and M. Zulkernine. 2008. Security Metrics for Source Code Structures. In *4th International Workshop on Software Engineering for Secure Systems*. ACM, 57–64.
[14] S. Demeyer. 2003. Maintainability versus Performance: What's the Effect of Introducing Polymorphism?
[15] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi. 2020. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology* 126 (2020), 106347.
[16] M. Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Educationl.
[18] S. Götz and M. Pukall. 2009. On Performance of Delegation in Java. In *2nd International Workshop on Hot Topics in Software Upgrades (HotSWUp '09)*. ACM, Article 3, 6 pages.
[19] S. Hayashi, M. Saeki, and M. Kurihara. 2006. Supporting refactoring activities using histories of program modification. *Transactions on Information and Systems* 89, 4 (2006), 1403–1412.
[20] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. de Lucena. 2015. Contrasting exception handling code across languages: An experience report involving 50 open source projects. In *IEEE 26th ISSRE*. 183–193.
[21] M. Kim, T. Zimmermann, and N. Nagappan. 2014. An empirical study of refactoring challenges and benefits at Microsoft. *TSE* 40, 7 (2014), 633–649.
[22] M. Lu and P. Liang. 2017. Automatic Classification of Non-Functional Requirements from Augmented App User Reviews. In *21st EASE*. ACM, 344–353.
[23] S. Moshtari and A. Sami. 2016. Evaluating and Comparing Complexity, Coupling and a New Proposed Set of Coupling Metrics in Cross-Project Vulnerability Prediction. In *31st Annual Symposium on Applied Computing*. ACM, 1415–1421.
[24] E. Murphy-Hill, C. Parnin, and A. P. Black. 2011. How we refactor, and how we know it. *TSE* 38, 1 (2011), 5–18.
[25] M. Paixao, M. Harman, Y. Zhang, and Y. Yu. 2017. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 394–414.
[26] C. Parnin and C. Görg. 2006. Lightweight visualizations for inspecting code smells. In *Symposium on Software visualization*. ACM, 171–172.
[27] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE TSE* 44, 6 (2018), 574–594.
[28] J. Ratzinger. 2007. *sPACE – Software Project Assessment in the Course of Evolution*. Doctoral Dissertation. Vienna University of Technology.
[29] Scientific Toolworks, Inc. 2020. *Understand*. https://scitools.com/support/metrics_list/?
[30] M. Siavvas and D. Kehagias, D.and Tzovaras. 2017. A Preliminary Study on the Relationship Among Software Metrics and Specific Vulnerability Types. In *International Conference on Computational Science and Computational Intelligence (CSCI)*. 916–921.
[31] N. Siegmund, M. Kuhlemann, M. Pukall, and S. Apel. 2010. Optimizing Non-functional Properties of Software Product Lines by means of Refactorings. In *4th International Workshop on Variability Modelling of Software-Intensive Systems*, Vol. 37. Universität Duisburg-Essen, 115–122.
[32] C. U. Smith and L. G. Williams. 2000. Software Performance AntiPatterns. In *2nd International Workshop on Software and Performance*.
[33] V. Soares, A. Oliveira, J. Pereira, A. C. Bibano, A. Garcia, P. R. Farah, S. Vergilio, M. Schots, C. Silva, D. Coutinho, D. Oliveira, and A Uchôa. 2020. *Website*. https://sbes2020refactoring.github.io/
[34] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Tenorio, M. Kim, and A. Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *17th ICSE*.
[35] D. Tenorio, A. C. Bibiano, and A. Garcia. 2019. On the customization of batch refactoring. In *3rd IWoR*. IEEE Press, 13–16.
[36] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *40th ICSE*. ACM, 483–494.
[37] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *34th ICSE*. IEEE Press, 233–243.