Why Do Developers Reject Refactorings in Open-Source Projects?

JEVGENIJA PANTIUCHINA, Software Institute, Università della Svizzera italiana
BIN LIN, Software Institute, Università della Svizzera italiana
FIORELLA ZAMPETTI, University of Sannio
MASSIMILIANO DI PENTA, University of Sannio
MICHELE LANZA, Software Institute, Università della Svizzera italiana
GABRIELE BAVOTA, Software Institute, Università della Svizzera italiana

Refactoring operations are behavior-preserving changes aimed at improving source code quality. While refactoring is largely considered a good practice, refactoring proposals in pull requests are often rejected after the code review. Understanding the reasons behind the rejection of refactoring contributions can shed light on how such contributions can be improved, essentially benefiting software quality.

This paper reports a study in which we manually coded rejection reasons inferred from 330 refactoring-related pull requests from 207 open-source Java projects. We surveyed 267 developers to assess their perceived prevalence of these identified rejection reasons, further complementing the reasons.

Our study resulted in a comprehensive taxonomy consisting of 26 refactoring-related rejection reasons and 21 process-related rejection reasons. The taxonomy, accompanied with representative examples and highlighted implications, provides developers with valuable insights on how to ponder and polish their refactoring contributions, and indicates a number of directions researchers can pursue toward better refactoring recommenders.

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

Additional Key Words and Phrases: Refactoring, empirical software engineering

ACM Reference Format:
Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. 2021. Why Do Developers Reject Refactorings in Open-Source Projects?. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 25 pages. https://doi.org/10.1145/1122445.1122456

Authors' addresses: Jevgenija Pantiuchina, jevgenija.pantiuchina@usi.ch, Software Institute, Università della Svizzera italiana, Lugano, Switzerland; Bin Lin, bin.lin@usi.ch, Software Institute, Università della Svizzera italiana, Lugano, Switzerland; Fiorella Zampetti, fiorellazampetti@gmail.com, University of Sannio, Benevento, Italy; Massimiliano Di Penta, dipenta@unisannio.it, University of Sannio, Benevento, Italy; Michele Lanza, michele.lanza@usi.ch, Software Institute, Università della Svizzera italiana, Lugano, Switzerland; Gabriele Bavota, gabriele.bavota@usi.ch, Software Institute, Università della Svizzera italiana, Lugano, Switzerland.

1049-331X/2021/1-ART1 $15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Refactorings are behavior-preserving changes focused on source code quality improvement [32]. As investigated in previous literature [60], there are many reasons to perform refactorings, for example, improving reusability, understandability, or testability. At the same time, refactoring has dark sides too. Although refactorings should be behavior-preserving, previous work has pointed out how refactoring can introduce bugs [24]. Additionally, developers may feel that it is overly risky to perform refactoring [40]. To mitigate such risks, refactoring-related changes can undergo a reviewing process before being accepted.

Code review can be performed using modern code review practices and tools: *e.g.,* Gerrit [9], the Pull Request (PR) review features available in GitHub, or even traditional code inspection meetings. As an outcome of a code review iteration, the reviewer can accept the proposed changes, reject them, or leave comments for the change author to address.

There are cases where the reasons for rejection are rather obvious, *e.g.,* test cases (or builds in general) that fail, the change causes a merge conflict, or, from a planning perspective, it is decided that it is not the right moment for refactoring. However, there can be reasons, more intrinsic to the design and implementation decisions made when performing the refactoring, that influence the rejection.

Pull Request (PR) acceptance has been widely studied [31][34][38][46][61][68]. There is also work that studied the refactoring process [50], proposed tools to aid refactoring review [33], pointed out refactoring risks [40] and negative effects [24].

To the best of our knowledge, the existing literature lacks systematic analysis of the reasons that lead to refactoring changes being rejected or of design and implementation decisions that developers tend to avoid upon refactoring source code.

Having such knowledge can help developers (which can be project members or occasional external contributors in the case of open-source projects) to better focus their refactoring efforts on changes that are useful and welcome, avoiding wasting time. Researchers can also exploit this knowledge to build better refactoring recommender systems [27] that avoid recommendations unlikely to be well-received by developers.

Our goal is to shed light on the reasons behind rejected refactoring contributions. We leverage two sources of information, *i.e.,* PR discussions on GitHub, and developers' opinions (from open-source and industry) collected through a survey.

To address this goal, first, we identified closed PRs from 2,057 Java open-source projects hosted on GitHub, selected using multiple criteria, such as a minimum number of commits and contributors, and currently active projects. From these projects, we sampled 951 candidate PRs from 395 systems, potentially contributing refactoring but closed without merging. Then, each PR was inspected by two independent annotators (from the paper authors) who either created tags describing the (inferred) reasons for which a PR was rejected or marked it as a false positive when it was not relevant for our study. After resolving annotation conflicts, we followed a card-sorting [63] approach to create, from such tags, 22 categories of refactoring-specific rejection reasons, and 12 rejection categories belonging to process-related reasons.

After that, we created an online survey aimed at (i) asking the perceived prevalence of the identified refactoring-specific rejection categories, and (ii) identifying refactoring rejection reasons we could have missed in our initial analysis. We distributed the survey to open-source and industrial developers, collecting a total of 267 responses.

Finally, through a further open coding of the additional comments provided in the survey, we improved our taxonomy, which in the end features 26 refactoring-related rejection reasons and 21 process-related reasons organized in a hierarchical taxonomy (depicted in Figure 3).[1]

The collected evidence can (i) serve as a guideline for developers to properly ponder and craft their refactoring contributions, and (ii) pave the way to better refactoring tools.

2 STUDY DESIGN

The *goal* of our study is to investigate the reasons why refactoring contributions are rejected. The *context* consists of (i) 951 manually-analyzed rejected PRs that contribute refactoring operations to 395 open-source Java projects hosted on GitHub and (ii) a survey performed with 267 developers. We focus on systems developed in Java to have a more cohesive set of projects to study. Also, all authors that have been involved in the manual analysis have a strong experience with Java, thus ensuring the needed expertise to understand the PRs. The study aims to answer the following research question:

What are the reasons for rejecting refactoring contributions?

We are interested in identifying the main reasons for which peer reviewers reject refactoring contributions submitted by developers through a pre-defined procedure *e.g.,* PRs on GitHub.

We answer our research question in two steps. First, we perform a mining-based study in which we manually analyze 951 rejected refactoring-related PRs to derive a taxonomy of reasons behind such rejections. Second, we survey 267 developers with the goal of (i) validating the previously defined taxonomy, *i.e.,* studying the prevalence of the identified rejection reasons for the perspective of software developers; and (ii) enriching our taxonomy with additional reasons not identified in the mining-based study. In the following, we describe the design of the mining-based study and of the developer survey.

2.1 Mining-Based Study

The Java software projects were selected from GitHub using the following selection criteria:

(1) *At least ten contributors and more than one year of history*, to exclude toy/personal projects.
(2) *At least 500 commits and 50 closed PRs*, to exclude projects having a small change history and that are unlikely to provide useful PRs for our analysis.
(3) *Modified at least once in the period from May 2019 to May 2020*, to filter out inactive projects. This criterion is also key for our survey, as we invite developers of the inspected projects to participate in our study.

We used the GitHub API for identifying the projects. Due to the limitations of the GitHub API, we first collected projects with a history of more than one year and having at least one commit between May 2019 and May 2020, which resulted in 10,175 projects. We then extracted the information related to #commits, #contributors, and #PRs for each project, and removed those not satisfying all our criteria.

Considering the expensive manual analysis that was to follow, we decided to focus on projects with at least 70 GitHub stars. This boundary produced 2,057 projects for our study, a number we consider sufficient for extracting rejected refactoring-related PRs. This only sets a lower bound rather than a star-based ranking which could bias our dataset with

[1]While for simplicity we use the phrasing "refactoring rejection reasons"/"process-related reasons", it is important to note that these reasons, especially the ones collected through the online survey, are *perceived* reasons.

artificially-popular projects [29]. The list of repositories, together with additional information (*e.g.,* number of commits, contributors, PRs, *etc.*) is available in our online appendix [16].

We used the GitHub API to extract PRs from the selected repositories. PRs on GitHub have the status "CLOSED", "MERGED", or "OPEN". To home in on the rejected contributions we only collected PRs with the status "CLOSED" and containing keyword "refactor" in the title or the description. This gave us 2,856,503 PRs. A manual inspection of several dozens of the identified PRs revealed that more than 80% PRs were false positives. Therefore, we applied extra filtering strategies to reduce false positives: We removed the PRs (1) not having keyword "refactor" in the title, (2) without comments from code reviewers, and (3) containing one of the following keywords in the last comment: "merged", "merging", "squashed", "superseded", "rebase". The last filter was added since we found that the GitHub APIs indicated as not merged several manually merged PRs. These PRs usually contain one of the listed keywords in their last comment. In the end, we collected 7,159 PRs.

While collecting the PRs, we only looked at the master/default branch of each project, as we were interested in identifying issues in the refactored code that did not make it suitable to be merged there; *i.e.,* a refactoring could be accepted in a branch, but may not be ready to be merged with the master yet.

2.1.1 Manual Analysis of Rejected Refactoring Discussions in Pull Requests. The 7,159 PRs obtained from our last step have been used for the qualitative analysis. While these PRs might still contain false positives (*i.e.,* they were merged manually or they are not related to refactoring), they are not a source of noise for our work as they will be discarded during the manual analysis.

Given the large amount of available PRs and limited human resources, we created a randomly-stratified sample of 951 PRs belonging to 395 projects. The strata represented the projects, *i.e.,* PRs were sampled across projects proportionally to the number of candidate PRs from the previous step. The total number of sampled PRs guarantees a significance interval (margin of error) of ±3% with a confidence level of 99%. The estimation has been performed applying a sample size (SS) calculation formula for an unknown population [58]:

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

and SS_{adj} for a known population pop:

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

where p is the predicted probability of the observation event to occur (we presume it is 0.5 if we do not know it a priori), Z_{α} is the value of the Z distribution for a given confidence level, and E is the estimated margin of error (±3%).

To create a taxonomy of refactoring rejection reasons, we followed a card sorting procedure. More specifically, according to Spencer [63], ours was a team work (*i.e.,* multiple people formed the theory collectively), open (no predefined categories), and remote (we used online tools from different locations) card sorting. In the first phase, PRs were labeled with tags (each PR was independently annotated by two people).

For this purpose, we created a Web App for tagging the selected PRs. The Web App (see Figure 1) shows to the annotator the PR title and the link to the discussion.

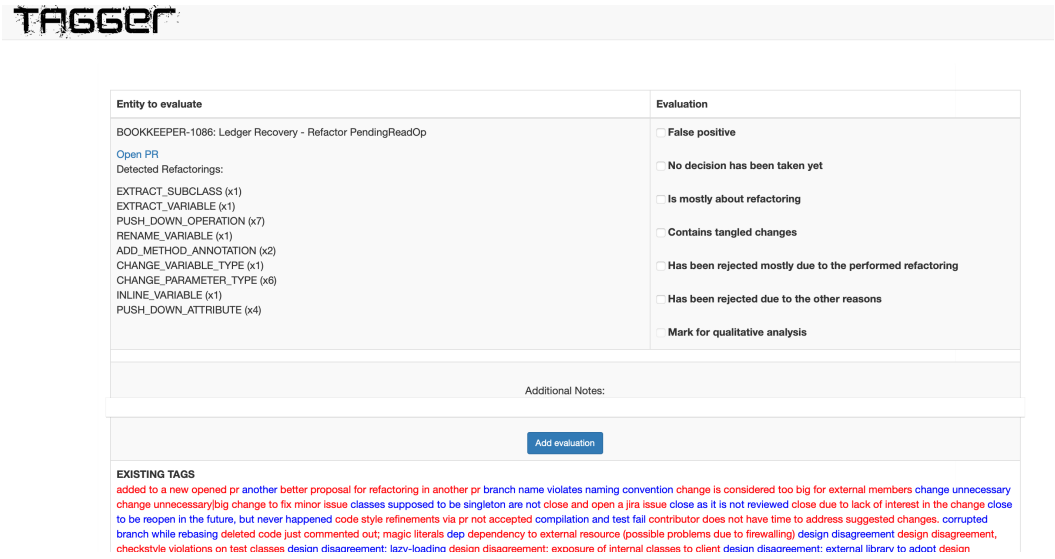

Fig. 1. Web App developed for the tagging process

The annotator has to verify whether: (i) the main goal of the PR was refactoring; (ii) the refactoring was tangled with other changes; (iii) the closed PR was merged manually; and (iv) the PR was rejected due to the refactoring or to other reasons.

If the PR is not related to refactoring or was merged, the annotator can tag it as "false positive". If the PR was rejected due to performed refactoring action(s), the annotator labels the PR using one or more existing tags or, if no existing tags fit, creates a new tag. The tag describes a reason for rejection inferred from the PR comments. During the tagging, the Web App shows the list of already created tags, which can be reused by an annotator when inspecting new PRs. In a card sorting strategy, this resembles the process of assigning a card to a group created by somebody else, and was done to avoid excessive growth of the number possible tags. Note that an evaluator did not know which tag was assigned to which PR nor who created it, but only that a tag is available. Thus, such a choice was meant to help annotators use consistent naming without introducing substantial bias.

Five authors took part in the annotation process. The developed Web App automatically assigned each PR to at least two of the involved annotators. A "warm-up" round was performed to help using consistent tags: after each annotator tagged ∼20 PRs, the five authors discussed the assigned tags and clarified good practices to use during the manual analysis. At the end of the tagging process, we collected a total of 249 unique tags, each one reporting a PR rejection reason. After the tagging process, the first author inspected all tags assigned to the PRs, merging very similar tags. Each conflict was resolved by a third annotator who was not involved in the tagging of that specific PR. A conflict arose when only one annotator believed the PR was rejected due to refactoring actions or when the two annotators used semantically different tags to express the reason for the refactoring rejection. The third annotator re-inspected the PR and took the decision based on personal judgment and the tags provided by the two original annotators.

After removing PRs tagged as false positives (585) and those with an unclear rejection reason (36), we obtained a set of 330 PRs from 207 projects. Such a sample ensures a

significance interval of ±5% with a confidence level of 95%. Our replication package [16] reports detailed information about the 207 projects, including: age in days and commits; size in terms of Java files, ELOC (Effective Lines of Code, excluding comments), and LOC (including comments); number of contributors in the change history; number of total PRs, of those having "refactor" in the title that have been closed, and of those having "refactor" in the title that were closed and merged; number of issues, forks, and watchers.

Overall, out of the 951 PRs we annotated, we had agreement for 660 of them, while the remaining 291 required a third annotator. We have followed a cooperative, open (*i.e.,* no predefined categories) card sorting, adopting the process defined by Spencer [63]: Since categories have not been defined beforehand, the computation of an inter-rater agreement is not foreseen, because the theory is being formed in such a circumstance. Also, several of our conflicts were due to different wordings expressing the same concept (*e.g., the PR is too large vs too many changes within a PR*).

2.1.2 Creating the First Taxonomy. During the second phase, five authors jointly worked on the available tags to conduct the card sorting [63] on the tags extracted from the 330 PRs. The goal was to further merge synonymous tags (*i.e.,* tags having the same meaning) and group tags into categories. The card sorting was conducted through an iterative process. First, each author sequentially inspected the tags. Then, after two iterations, all authors jointly discussed the conflicting cases in the categorizations they performed and assigned them to appropriate taxonomy categories, creating new ones when needed, and achieving consistency of category naming, until they felt there was no further improvement to be done. This process resulted in 22 refactoring-specific rejection reasons, and 12 process-related rejection reasons, *i.e.,* those not specifically-related to refactoring activities.

2.2 Survey with Developers

After creating the first taxonomy, we surveyed developers to validate the defined categories and to collect further rejection reasons we could not capture in our manual analysis. Since our focus is to study refactoring-specific reasons, and since other, process-related PR (or contribution) rejection reasons have also been investigated in previous work [31][34][38][61][68], we did not ask about the latter. We designed the survey organized in three sections according to guidelines from social science [35] and software engineering [41][42][43][44][56].

In the first section we asked for the demographic information, *i.e.,* the role played by the developer in her company/organization choosing between developer, software architect, technical lead, test analyst/tester/test engineer or specifying a different role; the years of programming experience in Java; and whether she had ever authored a refactoring contribution that was rejected or rejected one herself. If the participant answered "no" to the last two questions, the survey was interrupted (there was no previous experience with refactoring rejections).

In the second section, we asked participants to provide their agreement level to the statement "Refactoring contributions are often rejected because R_r" for each R_r (refactoring rejection reason) previously identified, using a 5-level Likert scale answer [52] (from *strongly agree* to *strongly disagree* with a *neutral* option) or a "I don't know" answer.

In the last section, respondents could provide, through two free-form text fields, (i) additional reasons for rejected refactorings not mentioned in our initial list, as well as (ii) characteristics that a PR implementing a refactoring must have for it to be merged.

We targeted developers through three different channels: (Ch1) mailing lists/forums of the projects in the mining study and personal emails to developers of the studied projects

directly involved in a rejected refactoring PR (as reviewers or contributors); (Ch2) personal knowledge; and (Ch3) social media and Reddit channels.

We collected answers for ten days, reaching 267 completed questionnaires. The roles of the respondents, and the channel through which we reached them are summarized in Table 1.

Table 1. Survey Respondents.

Role	Ch1	Ch2	Ch3	Total
Developer	138	16	6	160
Tech Lead	47	1		48
Architect	35	2	1	38
Other (Consultant, etc.)	18	1	2	21
Total	238	20	9	267

With respect to field experience of the 267 respondents, the whole spectrum is covered (88: 10+ years, 107: 3-10 years, 68: less than 3 years, 4 did not indicate it).

2.3 Taxonomy Refinement

Five authors ran an open coding procedure on the answers provided by the survey respondents, in particular to those related to the question in the third section of our survey, where we asked for additional reasons for rejected refactoring not mentioned in our initial taxonomy. After a first round of labeling by two independent coders, a third person reviewed them and assigned a final tag to each entry, defined new categories based on the new tags and added them to the taxonomy. After defining the categories, we started an iterative process to group similar categories or define new ones. We obtained 14 new categories for the rejection reasons (4 refactoring-specific and 9 process-specific), leading to our final taxonomy of 47 motivations (26 refactoring-specific and 21 process-specific) for rejecting refactoring contributions.

For the last open question in our survey, investigating the characteristics of an acceptable PR implementing a refactoring, an author went through all the answers to assign a set of tags to each answer. A different author went through all the defined tags to unify them.

2.4 Presentation and Discussion of the Taxonomy

We present our taxonomy, reporting interesting qualitative examples for each of its categories, highlighting those identified in the mining-based study and those complemented by our survey, and discussing implications. For the categories identified from PRs (for which we asked for the prevalence from the developers' point of view), we report results through diverging stacked bar charts.

2.5 Impact of Projects/Contributors Characteristics on the Taxonomy

Projects having different characteristics exhibit a different behavior in terms of accepting or rejecting refactoring PR. To investigate the extent to which this happens, we looked at how our taxonomy would change by only inspecting PRs extracted from GitHub projects having specific characteristics. We looked at projects having a different (i) number of contributors, (ii) history length measured in number of days between the first and the last commit documented on GitHub, and (iii) size, measured in Lines of Code (LOC), including code and comments.

Starting from the 207 projects that contributed at least one PR to our taxonomy, we split them into three groups, having a *low*, *medium*, and *high* number of contributors. The

thresholds to split the systems in the three groups have been obtained by analyzing the 33% and the 66% percentiles. Systems in the "*low*" group have less than 71 contributors (33% percentile), those in the *medium* have between 72 and 307 (66% percentile) contributors, and those in the *high* have more than 307 contributors.

Similarly, projects have been split into those having a (i) *short* (less than 2,009 days), *medium* (between 2,010 and 2,841 days), and *long* (more than 2,841 days) history; and (ii) *small* (less than 89,906 LOC), *medium* (between 89,907 and 246,303 LOC), and *large* (more than 246,303 LOC) size. It is important to understand that the categories we defined are relative to our sample of 207 projects that as we explained includes only projects having at least 10 contributors, 500 commits, and 50 PRs. Thus, our definition of *small* system, should not be seen in terms of absolute terms but relatively to our sample.

We tested whether, for the different root categories of the taxonomy, proportions of PRs vary across different buckets (*e.g.,* among small/medium/large projects, or projects having a short/medium/long history). We used proportion tests [51], a test suitable to compare multiple (> 2 proportions). We consider a significance level of 95%, *i.e.,* reject the null hypothesis of the test (H_0: proportions do not significantly differ) when the test produces a p-value< 0.05. Since for each variable the test is performed multiple times (one for each root category) we adjust p-values using the Benjamini-Hochberg correction procedure [28].

3 STUDY RESULTS

Figure 2 reports the developers' perception of the initial set of refactoring-specific rejection categories emerged from the PRs manual analysis. Percentages indicate negative (Disagree or Strongly disagree), neutral, and positive (Agree or Strongly agree) answers respectively.

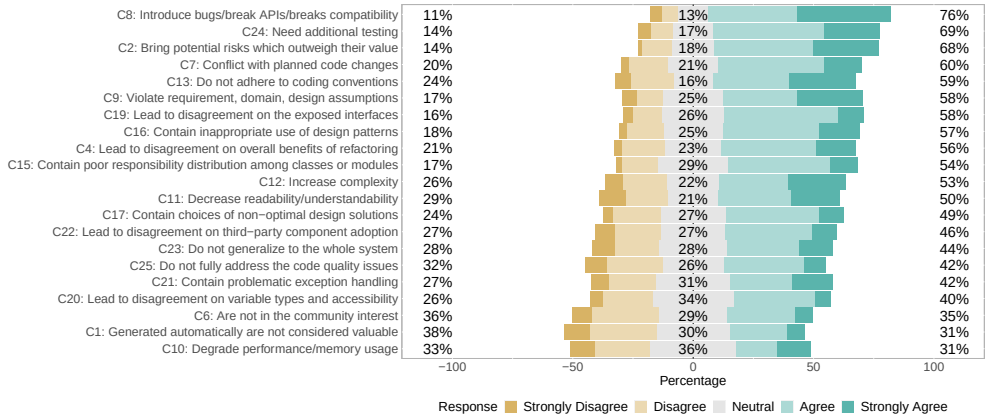

Fig. 2. Survey respondents' perception on the 23 refactoring rejection reasons identified from PRs

Figure 3 reports the taxonomy of refactoring rejection reasons. Such reasons can be either process-related (top-side) or specific to refactoring changes (bottom-side). The former are related to issues in the PR submission process (*i.e.,* the same code, submitted by avoiding those process-related issues, would likely be accepted), while refactoring-specific reasons are directly related to properties of the performed refactoring (*e.g.,* the refactoring contribution decreases code readability).

Categories colored in gray emerged from the PR-related study, and (for the refactoring-specific ones) were then assessed during the survey, whereas categories in green only emerged

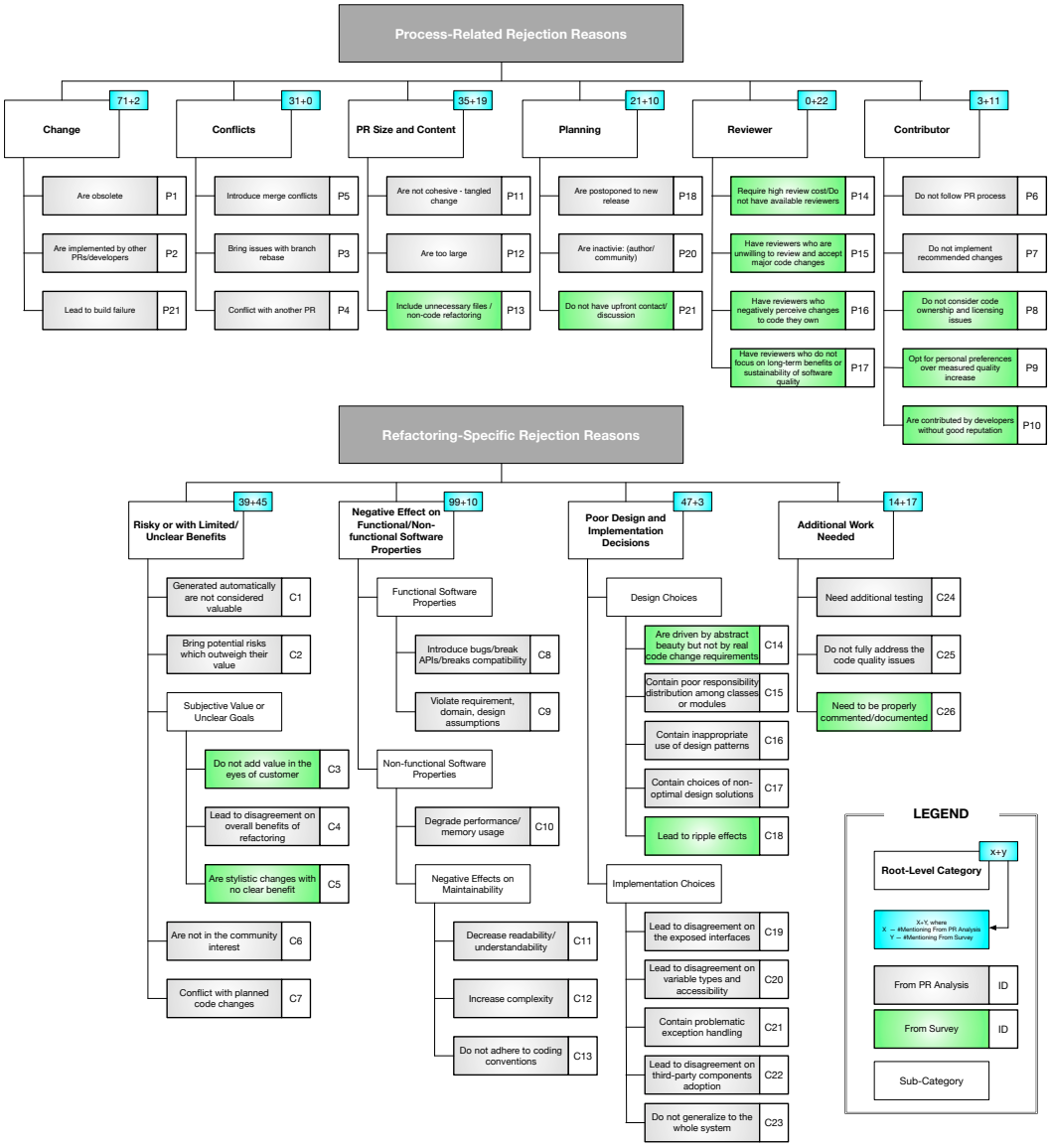

Fig. 3. Process-related and refactoring-specific rejection reasons

from the survey. The two numbers x and y in "x + y" attached to each root-level category R indicate the number of PRs that were rejected due to one of the reasons from PR analysis (x) and the number of surveyed developers that indicated one of the reasons in R as relevant for refactoring rejections they experienced (y).

In the following, we discuss our taxonomy, reporting interesting examples and outlining implications for researchers (indicated with the 💡 icon) and practitioners (🧑‍🔧 icon).

3.1 Process-Related Reasons

Process-related reasons for the rejection of refactoring contributions are depicted on the top of Figure 3. Since such reasons are not the main focus of our study, and since factors influencing PR acceptance have already been studied [31][34][38][61][68], we only briefly comment on them.

We identified five root-level categories in our manual analysis: (i) *Change* introduced in the PR, *e.g.,* due to refactoring operations performed on obsolete code; (ii) *Conflicts*, meaning cases in which the PR generated a code conflict (*e.g.,* a merge conflict); (iii) issues related to the *PR Size and Content*, in particular cases in which the PR is not cohesive and contains too many changes, do not have a clear focus; (iv) rejection reasons related to the PR *Planning*, such as postponing the refactored code to subsequent releases; and (v) reasons due to *Contributor* behavior (*e.g.,* does not implement changes required by the reviewers) or role (lack of reputation in the project).

We created a sixth category as a consequence of the survey we performed. Indeed, the surveyed developers also pointed to rejection reasons that are due to the *Reviewer*'s behavior during the code review process. This could not have been easily discovered by only looking at PR discussions, and points to possible subjectiveness in assessing the reason for why a PR is rejected. For example, consider the subcategory *Have reviewers who negatively perceive changes to code they own*. This subcategory accounts for cases where the owner of the refactored source code does not see the need for her code to be refactored. As it can be seen from Figure 3, besides the whole *Reviewer* root category, the survey also contributed with an additional five reasons (P13, P21, P8, P9, P10) to already existing root categories.

While not being the focus of our study, there are some lessons that can be drawn from the process-related reasons part of the taxonomy.

> When looking at related literature studying reasons impacting the PR acceptance, we see that process-related factors confirm findings reported in the study of Tsay *et al.* [68] and in the analysis of personality traits by Iyer *et al.* [38]. For instance, the lack of reputation for the contributor (*e.g.,* the contributor is a newcomer in the project) might decrease the chances of PR acceptance, as well as missing upfront contact/discussion (pointed out in the survey).

> Contributors and reviewers can derive from our taxonomy a set of good practices to apply in the PR process. On the contributor side, focusing on small cohesive PRs could help in increasing the chances of acceptance. Also, careful planning could avoid rejections related to obsolete contributions or changes that are work in progress in other PRs. *Communicating with the core developers sooner than later is important for contributors*. Looking at the reviewer's perspective, the study (and in particular the survey results) highlights the need to keep an open mind for changes to code they own and to consider the long-term benefits that refactoring contributions can bring.

3.2 Refactoring-Specific Reasons

3.2.1 Negative Effect on Functional/Non-functional Software Properties. This category of rejection reasons is related to the negative effects on software functional or non-functional (*e.g.,* performance, maintainability) properties.

Code contributions introduce bugs/break APIs/break compatibility (C8). While the main goal of refactoring changes is to improve the structure of source code without changing its behavior, sometimes developers unintentionally introduce bugs or API incompatibilities.

According to our survey, around 75% of the respondents (C8 in Figure 2) agree that refactoring contributions get rejected often due to this reason.

Example: In PR#2355 of the "`Spoon`" project [11], the contributor proposed to add the `final` qualifier to utility classes as advised by Java coding style guides, including a CheckStyle check [6]. However, while utility classes are not supposed to be instantiated or sub-classed, in practice, some developers extend them to easily import all their methods as a sort of API, that can be further extended in their project. Adding the `final` keyword would break client projects using such a mechanism and, for this reason, the PR was rejected.

> While it is natural to require refactoring contributions not to introduce bugs or alter APIs, this example indicates that contributors should also take precautions to avoid potential incompatibilities with client code, even when such code is not explicitly marked as an API.
>
> On the researchers' side, refactoring recommender systems should become *aware* of the cost of the recommended refactorings in terms of code changes. This point has also been raised by Hall *et al.* [36] who pointed to the code disruption caused by automated refactoring tools (*i.e.,* the need for updating a large part of the code base as a consequence of certain refactoring operations). Hall *et al.* focused on the disruption caused by the refactored code.

Our analysis shows that even small changes (*e.g.,* the addition of a `final` qualifier) can trigger a chain of code changes in the code base of a large number of client projects. Alerting developers when recommending such changes can improve the usability of refactoring recommenders and reduce the chance of accepting breaking changes.

Code contributions violate requirements, domain, or design assumptions (C9). When developers do not fully understand the rationale behind certain implementation choices made in a code base, they might submit refactoring contributions which are against the original project requirements or design. More than half (58%) of the developers claim that they often face contribution rejections due to this reason (C9 in Figure 2).

Example: In PR #3766 of the project `Druid` [2], the contributor extracted a new class from the `Granularity` class, while the goal of the maintainers was "*moving towards a single concept of Granularity*".

> This type of rejection can be avoided by providing detailed documentation about the maintainers' intentions together with a development roadmap.
>
> As reported by Aghajani *et al.* [20], documenting the rationale of code changes is considered a major "documentation issue" by practitioners, and techniques automating such a process can be of support. For example, creating techniques able to classify a code change as in need of documenting the rationale or not could support developers in such a task.

Code contributions bring negative effects on maintainability (C11, C12, C13). Another potential side effect of a refactoring action is that, besides its good intentions, it can decrease maintainability. This includes decreasing readability/understandability (C11 in Figure 2), increasing complexity (C12), and not adhering to coding conventions (C13). More than 50% of our respondents claim that they often face contribution rejections due to these reasons.

Example: In PR #2520 of the "`Spoon`" project [12], the contributor merged two `if` conditions into one return statement. While the expression became shorter, the original author of the code stated that he wrote a "longer implementation" because it was more readable for him.

This example sheds light on the subjectiveness of refactoring decisions. While shortening expressions make the code more concise, there are developers that find it more difficult to comprehend. On the contributors' side, this suggests that contacting core developers before implementing refactoring-related changes can avoid wasted effort.

On the researchers' side, instead, such a case supports the idea of consumer-related customization of recommender systems [22]. Specifically, in the context of refactoring, some developers (consumers of the recommendation) could appreciate certain refactoring recommendations because, for example, they are aligned with their programming style, while others may not accept it. Inferring the developer's programming style and "preferences" is an open research challenge that, if properly addressed, could substantially boost the usefulness of developer-related recommender systems.

Code contributions degrade performance/memory usage (C10). An undesired side effect of refactoring can be related to performance or memory usage degradation. This issue was indicated as reason for the rejection of refactoring contributions by 31% of our survey participants (C10 in Figure 2).

Example: In PR #403 of the `Apache Eagle` project [3], the developer proposed to replace low-level string manipulation functions (`indexOf`, `substring`) with regular expressions to improve the readability of the code used to parse log messages. The PR was rejected as the reviewer explained that replacing the current code with regular expressions was not an option, since they "*achieved more than 10 times performance gain by using string operations rather than reg-exp*" and that performance was highly important as in production there were issues with large amounts of audit logs.

While readability is appreciated, the adoption of less readable code is sometimes acceptable due to other non-functional requirements (*e.g.,* performance). Also in this case, to prevent developers from submitting unnecessary and performance-degrading refactoring contributions, the rationale for implementation choices could be documented in code comments. Another suggestion could be providing performance benchmarks and asking contributors to evaluate the performance change before submitting their code.

Looking at the problem from a researcher's perspective, state-of-the-art refactoring recommenders [26, 66] ignore the heterogeneity of modern software, and the different priorities that non-functional requirements may have in different contexts. Future work should consider integrating into these recommender systems the possibility to define a priority list of non-functional properties that developers are (or not) willing to sacrifice when applying a refactoring. This would allow generating more meaningful refactoring recommendations.

3.2.2 Additional work needs to be performed. This category of rejection reasons is related to insufficient work presented in the refactoring contribution, and extra tasks that should be performed to get the contribution accepted.

Code contributions need additional testing (C24). This is one of the most common issues developers encountered in this category, agreed by 76% of developers (C24 in Figure 2). The required test can be either functional or non-functional.

Example: In PR #2658 of the `Micronaut` project [15], the contributor was asked to evaluate the performance change after refactoring, while in other PRs (*e.g.,* PR #611 of the `grakn` project [10]) more functional testing was required.

> While the availability of tests is important for PR acceptance [69], it is particularly important for refactoring changes, which should not change behavior. Leveraging continuous integration infrastructures could help to achieve this goal, not only by ensuring that refactoring does not break changes, but also by checking that refactored code is properly tested (*e.g.,* by measuring coverage) and by checking there are no regression in non-functional properties such as performance/energy [59] or security [48].

Code contributions do not fully address the code quality issues (C25). Refactoring contributions can be rejected because further refactoring is needed to fully address the quality issue they target. 42% of the survey participants agree that this is a frequent reason for refactoring rejection (C26 in Figure 2).

Example: In PR #2393 of the `Open Event Android` project [8], the contributor performed extract class refactoring on a component responsible for too many tasks. The reviewer suggested that the extracted classes could be further refactored into even more fine-grained classes.

> This example shows the subjectivity and iterative nature of refactoring. From a refactoring recommender perspective, including developer feedback in the automated generation of refactoring solutions [23, 37] can help in reaching both these objectives by generating solutions that are (i) well-suited for the specific developer, and (ii) further refined when needed.

Code contributions need to be properly commented/documented (C26). Some survey responses mentioned the lack of proper comments/documentation as a reason for refactoring rejections. This suggests that developers should invest more time in non-coding activities during refactoring to make their contribution easier to understand. Also, automatic documentation techniques [57] may be used to propose to the developer code comments to be injected while refactoring.

3.2.3 Risky Or with Limited/Unclear Benefits. This category of rejection reasons is related to potential risks refactoring contributions bring or limited/unclear benefits they can provide.

Refactorings generated automatically are not considered valuable (C1). Sometimes contributors use tools to automatically generate refactorings, which are not always considered valuable. One third of the respondents (31%) claim this as possible reason for refactoring contribution rejection (C1 in Figure 2).

Example: In PR #286 of the `JITWatch` project [1], the maintainer clearly stated that he does not accept PRs generated by refactoring tools, and suggested the contributor to rather pick an open issue to help improve the project.

> In this case the reason for the rejection is not the quality of the proposed refactoring but rather the fact that there is no developer backing up and explaining the value of the proposed refactoring. While automated refactoring can provide useful support, developers should have "the last word" and properly adapt suggested changes. Automatically explaining the rationale for a recommended refactoring, while being an interesting research direction, may be insufficient.

The potential risk of refactoring contributions outweighs their value (C2). One frequently encountered issue with refactoring contributions is that their value might be outweighed by the risks they bring, agreed on by 68% of our survey participants (C2 in Figure 2).

Example: In PR #88 of the `Universal Media Server` project [18], the reviewers rejected the PR as they believe that "*it's better to have stable code that is poorly-written than potentially make it unstable but well-written*", and are "*against rewriting code if there is no other gain than that 'it looks better' since any change is bound to introduce new bugs*".

> The risks of refactoring operations are known in the literature, including the introduction of bugs [24]. Even IDE-integrated refactoring engines have been shown to introduce bugs in software under specific circumstances [62]. Regression testing of the refactoring contributions should always be performed to ensure the absence of behavioral changes.

Code contributions have subjective value or unclear goals (C3, C4, C5). The benefit of some refactoring contributions can be minor or unclear, and maintainers might not bother to merge the code. 56% of our survey responses indicated that this is often the reason for refactoring rejection (C4 in Figure 2).

In PR #211 of the `EssentialsX` project [7], the contributor refactored the code to include Java 7 features. He thought that the commit "*would help to update the code to take advantage of Java 7 and to help keep the code clean and up-to-date with the latest Java conventions*", and mentioned that another contributor agreed with the implemented changes. However, two reviewers commented about their perceived lack of value for the implemented changes, closing the PR.

> Such an example stresses the point about the subjectiveness of refactoring choices. In our survey, some respondents also mentioned that refactoring contributions are likely to be rejected when they are merely stylistic changes without clear benefits or when they do not add extra value to customers.

Code contributions are not in the community interest (C6). Sometimes refactoring contributions might bring changes not in line with the community. This reason is agreed on by one third (35%) of the respondents (C6 in Figure 2).

Example: In PR #2524 of the `XChange` project [14] the contributor proposed to simplify several central abstractions to pure interfaces. As reported by the reviewer, those changes are done to "*peruse a totally different way of*" implementing modules "*without an overall consensus*" of the community.

> While contributors might believe there are better ways of implementing code in a given project, looking for consensus of at least, of the core developers of the project is fundamental for all code changes and, in particular, for refactoring contributions. Indeed, appreciating the value of refactoring is subjective.

Code contributions conflict with planned code changes (C7). Another common reason for rejecting refactoring contributions, indicated by 60% of the survey participants (C7 in Figure 2), are conflicts they generate with ongoing or planned code changes.

Example: PR #645 of the `ExoMedia` project [5] was rejected as the refactored code in this Android app was going to be migrated to Kotlin.

> It is not uncommon to see contributions conflict with project plans. To prevent this type of effort waste, maintainers might consider adding relevant comments for the part of code which is or will be under significant changes, and to make it clear and publicly available the project road-map.

3.2.4 Poor Design And Implementation Decisions. This category is related to the disagreement between contributors and reviewers on design and/or implementation choices.

Design choices. The reasons for refactoring rejections reflected in our survey include "poor responsibility distribution among classes or modules (C15)", "inappropriate use of design patterns (C16)", and "choice of non-optimal design solutions (C17)". Around half of the survey participants agree that these are common refactoring contribution rejection reasons.

Example ("inappropriate use of design patterns"): In PR #123 of the `MonkeyEngine` project [13] the contributor applied a strategy pattern and introduced some additional abstractions to source code, and after a lengthy discussion, the reviewers decided to close the PR as they consider the PR was too *"invasive"*. While abstraction can enhance the extensibility and reusability of source code, it can also increase complexity.

Example ("poor responsibility distribution among classes or modules"): In PR #846 of the `XWiki Platform` project [19], the maintainer disagreed on how the contributor organized the structure of tests.

Example ("non-optimal design solutions"): In PR #155 in the `Spring Cloud Stream` project [17], the reviewers came up with a better refactoring idea after some iterations, and decided to implement the new idea in a fresh PR.

> Literature provides controversial suggestions on the use (and abuse) of design patterns: some studies suggest that they should always be preferred to simpler solutions because of their flexibility [45], whereas others indicate that they can make source code clumsy, might harm software evolution [39], and degrade performance [72]. Contributors should think about benefits for future changes, but, also, consider negative effects on understandability, maintainability, and performance.

Respondents also expressed concerns that refactorings are often driven by abstract beauty but not by real code change requirements. Refactorings have ripple effects, *i.e.,* they often lead to other tasks. Contributors should consider the extra value of refactoring in addition to improving the readability.

> Contributors should keep the refactoring self-contained and possibly well-focused, such that the overall system will not be dramatically impacted.

Implementation choices (C19, C20, C21, C22, C23). The reasons for refactoring rejections in our survey include "disagreement on the exposed interfaces (C19)", "disagreement on variable types and accessibility (C20)", "problematic exception handling (C21)", "disagreement on third-party components adoption (C22)", and "being unable to generalize to the whole system (C23)".

Around 40% of the survey participants have experienced refactoring rejections due to C20-C23. 58% of developers claim "disagreement on the exposed interfaces" was often the reason for refactoring rejection.

Example: In PR #1380 in the `Pulsar` project [4] the reviewer and the contributor disagreed on which getter/setter methods should be kept/removed.

> The reasons related to implementation choices stem from overly narrow views of the system upon performing a refactoring. Access to system-wide documentation and broad knowledge of the overall system's design may be required even when refactoring actions are focused.

3.3 How Projects/Contributors Characteristics Impact our Taxonomy

Table 2 reports the achieved results for size, history length, and number of contributors.

Table 2. Number of PRs when considering projects having different sizes measured in lines of code

Root Category	Taxonomy	Small	Medium	Large	Short	Medium	Long	Low	Medium	High
		LOC			History Length			Contributors		
Additional work needs to be performed	14	6%	4%	4%	7%	4%	2%	4%	5%	3%
Change	71	16%	22%	27%	24%	16%	2%5	23%	23%	15%
Conflicts	31	12%	9%	7%	14%	7%	7%	13%	10%	3%
Negative eff. on func./non-func. prop.	99	30%	29%	30%	4%	10%	5%	33%	28%	31%
Planning	21	5%	10%	4%	14%	14%	16%	2%	7%	12%
Poor design and implem. decisions	47	17%	11%	15%	14%	14%	16%	14%	16%	10%
PR size and content	35	16%	9%	7%	11%	7%	14%	13%	10%	10%
Risky or with limited/unclear benefits	39	12%	10%	13%	8%	16%	13%	11%	11%	15%
Unique PRs	327	109	108	110	115	109	103	94	176	57

The table shows for each root category in our taxonomy the number of PRs that contribute to it in the entire taxonomy (column "Taxonomy", which considers all 207 systems as a whole). Note that we only considered root categories to make sure we have enough data for such an analysis. Also, we excluded the "Contributor" category since it had too few PRs contributing to it (3), and the "Reviewer" category since it emerged as output of the survey rather than of the PR study. The last row shows the number of unique PRs in our taxonomy contributed by systems having different characteristics (*e.g.,* 109 PRs come from *small* systems). Finally, the percentages in the table show the percentage of PRs from specific types of systems (*e.g., small* systems in Table 2) that belong to a specific root category. For example, 17 PRs coming from *small* systems contribute to the *Change* root category (16%).

Since a PR can belong to more than one root category, the number of unique PRs is not the sum of the PRs in each category, and therefore the percentages do not add up to 100%.

There are no major differences in how the different "categories" of projects contribute to the overall taxonomy. This is confirmed by the proportion test, which never highlighted statistically significant differences in proportions of PRs among different categories (p-values always greater than 0.05). Such a finding is valid for all categories and it is likely due to the fact that, as explained, our selection criteria for the subject systems probably created a *cohesive* set of projects. More interesting are the findings in Table 3, in which we analyze how PRs opened by developers having different levels of experience within each of the 207 projects contribute to our taxonomy.

Table 3. Number of PRs in each root category when considering PRs authored by contributors with different experience (# past commits in the system).

Root Category	Taxonomy	Q_1	Q_2	Q_3	Q_4
Additional work needs to be performed	14	4%	3%	0%	7 %
Change	71	20%	22%	25 %	34%
Conflicts	31	10%	8%	17%	7 %
Negative eff. on func./non-func. prop.	99	29%	33 %	50%	28%
Planning	21	8 %	6 %	0 %	0%
Poor design and implementation decisions	47	14%	22 %	8 %	7 %
PR size and content	35	10%	11 %	8 %	17 %
Risky or with limited/unclear benefits	39	13%	6%	17%	7%
Unique PRs	327	250	36	12	29

Table 3 has the same format of the previous tables. However, we split the developers into four categories based on the quartiles. The first quartile Q_1 groups developers that, within a project, had an experience (measured as number of past commits they performed before opening the PR we analyzed) lower than 25% of the developers of the system. These are basically the developers with the lowest experience in a specific project. Similarly, Q_2 are the developers falling in the 25% to 50% percentile, Q_3 in the 50% to 75% percentile, and Q_4 above the 75%. We decided for a more fine-grained analysis (four categories rather than three as done for the projects' characteristics) to achieve a stronger distinction between those that could be considered almost newcomers (Q_1) and very experienced developers (Q_4).

The results in Table 3 show an interesting finding: developers in the 25% percentile are responsible for 76% of the PRs rejected due to issues documented in our taxonomy. Among the other three categories, there is not a strong distinction. As previously discussed, this confirms previous findings in the literature, reporting the lack of experience of the contributor as one of the reasons for the rejection of PRs. Also, this indicates that our taxonomy is particularly relevant for developers moving their first steps in an open-source project.

4 THREATS TO VALIDITY

Construct validity threats concern the relationship between theory and observation. The first part of our study focused on PRs that were mostly about refactoring. As pointed out by Murphy-Hill *et al.* [50], floss refactorings (performed together/as a consequence of another change) tend to be more frequent than pure ("root-canal") refactorings. Nevertheless, during the manual analysis we also checked whether the refactoring was performed together/as a consequence of another change, and this happened for 172 out of 330 PRs, *i.e.,* 50.7%, which is in line with the results reported by Murphy-Hill *et al.* [50]. Also, validating the initial taxonomy and collecting further refactoring rejection reasons through the survey helps to avoid the root-canal refactoring bias. Another threat is caused by the fact that rejection reasons originating from the survey are mostly personal opinions/perceptions. That is, respondents could have other reasons for rejecting refactorings (even including personal conflicts with other developers) they do not want to admit. We mitigated this threat by highlighting these categories in the taxonomy, distinguishing from those inferred from the PR manual analysis.

The taxonomy defined in our study mostly derives from a manual analysis of rejected refactoring-related PRs. However, we did not look into *accepted* PRs to assess the extent to which the same "reasons" we documented in our taxonomy are present there. In other words, it is possible that, while we found many instances of PRs rejected due to reason R, there are many other PRs in which R was not considered a reason to reject the refactoring contribution.

Finally, our study is based on the idea that refactoring-related PRs are rejected but we did not study how often this happens in practice (but only *why* this happens). This is due to the fact that, as shown in our manual analysis, automatically identifying refactoring-related PRs is challenging and, on top of that, ensuring that a PR has been rejected **due** to refactoring-related reasons requires a manual inspection of the PR. However, to get an idea of the magnitude of the phenomenon, we extracted from the set of 207 GitHub projects contributing at least one PR to our taxonomy, the number of closed PRs having "refactor" in the title. We found 26,015 instances, of which 7,481 (29%) while closed have not been merged (thus possibly being rejected). This supports the relevance of the phenomenon investigated in our study.

Internal validity threats concern factors internal to our study that could have influenced our results, and are mainly related to possible subjectiveness and imprecision when creating the taxonomy. To mitigate this threat, we used a multiple tagger approach in the manual analysis of PRs and of the open answers provided by developers. Note that, being this an open card sorting process where the categories are not clearly predetermined, we did not aim at reaching an agreement already during the tagging phase. This is why a conflict resolution phase was embedded in our process.

As explained in Section 2.5, we analyze the extent to which the characteristics of the analyzed projects, as well as of their contributors, could impact on our results, *i.e.,* on the distribution of PRs for the different refactoring rejection reasons. However, as shown in Section 3.3, we did not find any statistically significant difference.

It is still possible that refactoring rejection reasons are our interpretation of what is reported in the PR discussion or mentioned in the survey answers. Moreover, to create the taxonomy, we followed a card sorting approach [63] involving multiple authors, with iterative refinements performed until no more changes were done by anybody.

During the tagging the web app we employed showed the list of already created tags to avoid an excessive growth of the number of possible tags. A possible consequence is that annotators avoided the creation of a new tag when an already existing tag was "close enough" to the one they wanted to define (*i.e.,* annotators maximized the reuse of existing tags).

External validity threats concern the generalizability of our findings. Our study focuses on Java projects. Thus, our findings are specific to Java, since other languages may trigger specific refactoring opportunities we did not consider. Furthermore, the first part of the analysis has focused on open-source projects hosted on GitHub, where refactoring contributions are reviewed as PRs. We mitigate this threat through the survey which also involves respondents working in industry.

While we manually analyzed 951 PRs, our final taxonomy stems from 330 manually identified true positives, coming from 207 repositories, leading to a small average number of PRs per repository (1.6). This affects the representativeness of our taxonomy.

5 RELATED WORK

We first briefly discuss related work on pull request acceptance, and then focus on studies investigating why and how developers perform refactoring, and the challenges developers encounter during refactoring.

5.1 Studies on PR acceptance

Gousios *et al.* [34] conducted an exploratory study revealing how metrics measuring the size of a PR have an impact on its acceptance, highlighting reasons for PR rejection. They found that only 13% of PRs are rejected due to technical issues, while 53% of the rejections are due to the distributed nature of the PR process. Our study confirms that, also for refactoring, PR size plays a major role, although there are several other technical factors also impacting acceptance.

Lenarduzzi *et al.* [46] investigated whether code quality (assessed in terms of code smells, anti-patterns and coding style violations) matters when accepting a PR. While their results claim that code quality does not affect PR acceptance at all, our study highlighted how PRs were rejected due to code quality issues and to the lack of adherence to coding standards. A possible reason for such a divergence is that Lenarduzzi *et al.* detected themselves smells and violations (which may or may not be relevant for developers), whereas we found discussions highlighting violations of projects' coding style guidelines.

Other authors looked at social aspects influencing pull request acceptance.

Tsay *et al.* [68] looked at both technical and social factors influencing PR acceptance. Their findings point out how the lack of reputation of the contributor (*e.g.,* the contributor is a newcomer in the project) reduces the chances of PR acceptance, as well as missing upfront contact/discussion. Chen *et al.* [31] showed how support (or on the contrary disapproval) and the availability of alternate solutions contribute to drive PR acceptance. Soares *et al.* [61] found that technical (*e.g.,* programming language, number of commits) and social factors (*e.g.,* whether the submitter is a newcomer or external contributor) have an impact on PR acceptance. Iyer *et al.* [38] found that diversity in personality adds value to open-source projects. Their results point out that the likelihood of PR acceptance is influenced by personality traits of developers, both submitters and closers. For instance, submitters having high openness, conscientiousness and low extroversion have more chances for PR acceptance.

Differently from the above studies, we analyze refactoring-specific factors with the aim of identifying the main reasons for not accepting PRs implementing a refactoring change.

5.2 Studies on refactoring activities, risks and challenges

Refactoring has been studied in the literature from many different perspectives. These include the probability of introducing bugs while refactoring [24], merge conflicts caused by refactorings [47], how developers' productivity is impacted by refactoring [49], and how quality indicators change as consequence of refactoring actions [21][30][64][65].

In a field study with 328 Microsoft engineers, Kim *et al.* [40] investigated when and how developers refactor code. They identified refactoring risks, which can be mapped to reasons for rejection. These include, in order of importance, regression bugs/build breaks, testing cost, time taken from other tasks, merge conflicts, difficulty to code review and churns. Most risks map to our process-related reasons, breaking behavior, and need for better testing. Our work goes further, identifying 26 refactoring-specific reasons, and expanding on the process-related reasons (21 in total).

Murphy-Hill *et al.* [50] investigated how developers perform refactorings. The authors analyzed refactoring operations in eight datasets. One of the studied datasets contains usage data from 41 developers using the Eclipse IDE capturing an average of 66 hours of development time per developer.

Interesting findings are that (i) programmers rarely configure refactoring tools; (ii) commit messages do not help in predicting refactoring, since developers do not explicitly report their refactoring activities in them; (iii) developers often interleave refactoring with other programming activities; and (iv) most refactoring operations (∼90%) are performed without the help of any tools. As Murphy-Hill *et al.* [50] observed, refactoring operations occur frequently together with/in consequence of other changes ("floss refactoring"), as opposed to pure ("root-canal") refactoring. In our study we carefully determined, while reviewing PRs, whether a rejection was due to refactoring or to other reasons.

Wang *et al.* [71] studied the factors motivating refactoring operations. The authors built an empirical model featuring 12 *intrinsic* (when refactoring is initiated without any obvious external reward, *e.g., Self Esteem*) and external motivators (*e.g., Recognitions from Others*). Related to our work, it is possible that some intrinsic factors could let developers initiate refactoring actions which end up not having a clear value or having other weaknesses which determine their rejection.

Silva *et al.* [60] focused on refactoring operations mined by using RMiner [67]. The authors contacted contributors of the mined refactorings asking for the motivations behind the performed changes. They defined a catalog of 44 motivations for 12 types of refactoring

operations. Our work is not specific to refactoring types, but aims at identifying points to ponder before proposing a refactoring action, whatever type it is.

Paixao *et al.* [53] investigated developers' intents when performing refactoring actions and their sequential evolution during code review. A similar study was presented by Pantiuchina *et al.* [54], in which the authors qualitatively analyzed refactorings in merged PRs with the goal of complementing refactoring motivations already known in the literature [40, 60, 71]. Moreover, the authors performed a quantitative analysis to study which process- and product-related factors correlate with refactoring actions.

Peruma *et al.* [55] studied factors motivating developers to perform refactoring actions, focusing on rename refactorings. Their findings show that in most cases rename refactoring is applied to narrow the identifier meaning. This finding was confirmed by Pantiuchina *et al.* [54]. We focus on reasons for refactoring rejection rather than on refactoring rationale. At the same time, we found that the lack of a clear goal and not properly-documented refactoring are causes for rejection.

Bavota *et al.* [25] studied the relationship between code smells, metrics, and refactoring by detecting refactorings between two subsequent releases. Their findings revealed that there is no causation between code having a smell or increased complexity and subsequent refactoring of the code. Vassallo *et al.* [70] quantitatively investigated factors correlating with refactorings. They considered factors related to why, when, and by whom refactoring was performed.

As in our work, the importance of code review in the context of refactoring was highlighted by Ge *et al.* [33]. They proposed ReviewFactor, a tool to highlight refactoring-related changes during code review. Our work further highlights the importance of tools like ReviewFactor, and suggests that further support is required to help developers on deciding when performing (or not) a refactoring, and how to review it.

6 CONCLUSION

When Fowler published his seminal book on Refactoring, he promoted the idea that one does not just first design and then code, but rather that source code is malleable, and being constantly refactored to reflect an ever-changing environment, to better adhere to sound design principles and improve its quality [32].

We investigated the reasons –as they can be inferred from pull requests (PRs) or as they are reported by developers– for rejecting refactoring contributions. We manually analyzed 951 pull requests (PRs) from 395 Java projects and complemented our findings through a large survey with 267 developers. As a result, we defined a taxonomy of refactoring rejection reasons, divided into process-related and refactoring-specific reasons.

Besides the reasons related to behavior-breaking changes, we found that refactoring contributions can be rejected due to the lack of clear benefits, or for risks that outweigh the value of the proposed change. Also, the lack of proper knowledge of a project's requirement/design assumptions, of its current development planning, or its coding styles can contribute to a negative outcome of proposed refactorings. Important factors, as survey respondents pointed out, are the lack of a preliminary discussion of refactoring intentions and the limited (or absent) description of the proposed changes and their goals/rationale.

The collected evidence can help to define guidelines for developers. Also, it can serve as the basis for approaches and tools able to help developers to craft sound and acceptable refactoring contributions (*e.g.,* by applying the correct coding style and by generating appropriate change descriptions), provide a quantitative assessment of their risk/benefits, and predict whether they are likely to be accepted.

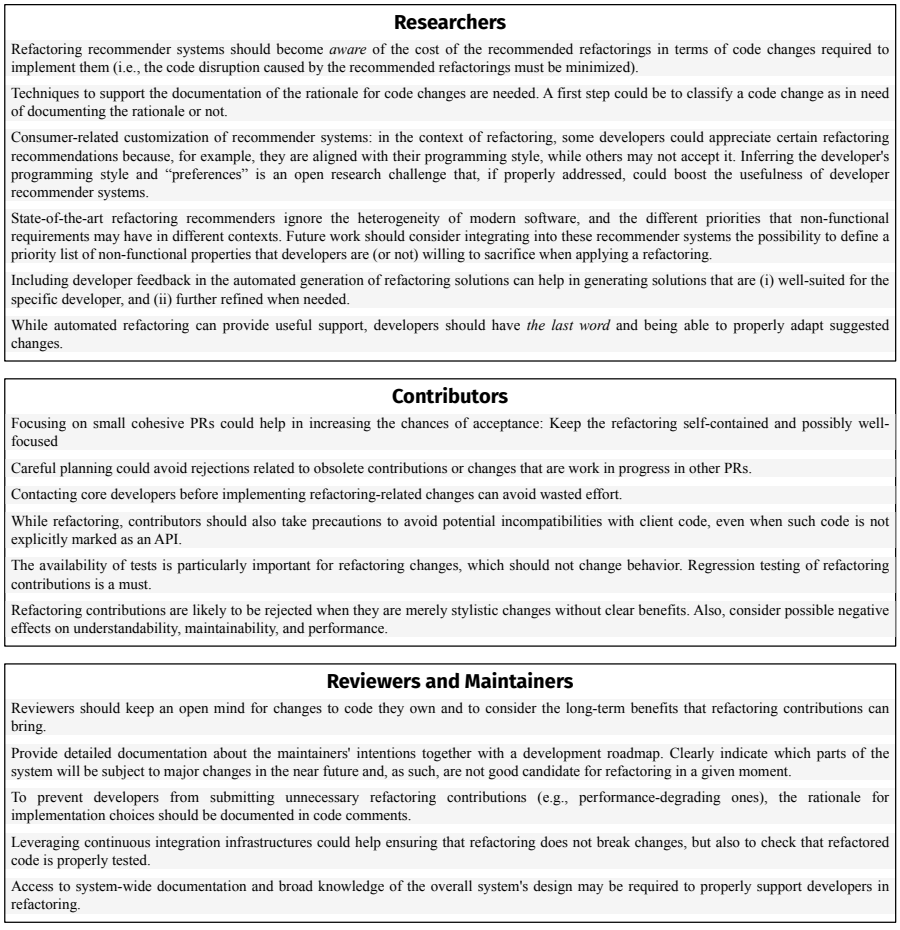

Fig. 4. Summary of lessons learned from our study

Figure 4 summarizes the main take-away messages derived from our study. In the top part we report practical implications for researchers, that could help in designing better refactoring recommender systems. While most of these implications are linked to refactoring recommenders, several of them can be seen as more general for developers' recommender systems. The middle part concerns implications for developers authoring refactoring contributions. The final goal of these guidelines is to increase the chance of acceptance and contributing refactorings that are appreciated by reviewers and maintainers of open-source projects. Finally, the bottom concerns (i) reviewers of refactoring-related PRs, and (ii) maintainers of open source projects. Following the provided guidelines could help in better guiding developers in contributing refactoring changes and avoiding wasted refactoring effort.

Our future work lies in leveraging the taxonomy we distilled to design and develop new refactoring recommender systems, following the points highlighted in Figure 4.

7 DATA AVAILABILITY

The complete study material and data are available in our replication package [16].

ACKNOWLEDGMENTS

We are grateful for the financial support by the Swiss National Science foundation through SNF Project JITRA (172479) and SENSOR (183587).

REFERENCES

[1] AdoptOpenJDK/jitwatch/ Pull Request #286. https://github.com/AdoptOpenJDK/jitwatch/pull/286 (Last access: 08/10/2020).
[2] apache/druid Pull Request #3766. https://github.com/apache/druid/pull/3766 (Last access: 08/10/2020).
[3] apache/eagle/ Pull Request #403. https://github.com/apache/eagle/pull/403 (Last access: 08/10/2020).
[4] apache/pulsar/ Pull Request #1380. https://github.com/apache/pulsar/pull/1380 (Last access: 08/10/2020).
[5] brianwernick/ExoMedia/ Pull Request #645. https://github.com/brianwernick/ExoMedia/pull/645 (Last access: 08/10/2020).
[6] CheckStyle. http://checkstyle.sourceforge.net/ (Last access: 08/10/2020).
[7] EssentialsX/Essentials/ Pull Request #211. https://github.com/EssentialsX/Essentials/pull/211 (Last access: 08/10/2020).
[8] fossasia/open-event-droidgen/ Pull Request #7. https://github.com/fossasia/open-event-droidgen/pull/2393 (Last access: 08/10/2020).
[9] Gerrit. https://www.gerritcodereview.com.
[10] graknlabs/grakn Pull Request #611. https://github.com/graknlabs/grakn/pull/611 (Last access: 08/10/2020).
[11] INRIA/spoon/ Pull Request #2355. https://github.com/INRIA/spoon/pull/2355 (Last access: 08/10/2020).
[12] INRIA/spoon/ Pull Request #2520. https://github.com/INRIA/spoon/pull/2520 (Last access: 08/10/2020).
[13] knowm/XChange/ Pull Request #123. https://github.com/jMonkeyEngine/jmonkeyengine/pull/123 (Last access: 08/10/2020).
[14] knowm/XChange/ Pull Request #2524. https://github.com/knowm/XChange/pull/2524 (Last access: 08/10/2020).
[15] micronaut-projects/micronaut-core/ Pull Request #2658. https://github.com/micronaut-projects/micronaut-core/pull/2658 (Last access: 08/10/2020).
[16] Replication Package https://github.com/replicatio/package.
[17] spring-cloud/spring-cloud-stream/ Pull Request #155. https://github.com/spring-cloud/spring-cloud-stream/pull/155 (Last access: 08/10/2020).
[18] /UniversalMediaServer/UniversalMediaServer/ Pull Request #88. https://github.com/UniversalMediaServer/UniversalMediaServer/pull/88 (Last access: 08/10/2020).
[19] xwiki/xwiki-platform/ Pull Request #846. https://github.com/xwiki/xwiki-platform/pull/846 (Last access: 08/10/2020).
[20] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David Shepherd. Software Documentation: The Practitioners' Perspective. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*.
[21] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and Software Technology* 51, 9 (2009), 1319 – 1326.
[22] Gabriele Bavota. Mining Unstructured Data in Software Repositories: Current and Future Trends. In *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*. 1–12.
[23] Gabriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization. In *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings*. 75–89.
[24] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When Does a Refactoring Induce Bugs? An Empirical Study. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 104–113.

[25] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1 – 14.
[26] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering* (2013), 1–48.
[27] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Recommending Refactoring Operations in Large Software Systems. In *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, 387–419.
[28] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.
[29] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129.
[30] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. 74–83.
[31] Di Chen, Kathryn T. Stolee, and Tim Menzies. Replication can improve prior results: a GitHub study of pull request acceptance. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. 179–190.
[32] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[33] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson R. Murphy-Hill. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*. 71–79.
[34] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 345–355.
[35] Robert M. Groves, Floyd J. Jr. Fowler, Mick P. Couyper, James M. Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey Methodology, 2nd edition*. Wiley.
[36] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn. Establishing the Source Code Disruption Caused by Automated Remodularisation Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 466–470.
[37] Mathew Hall, Neil Walkinshaw, and Phil McMinn. Supervised software modularisation. In *28th IEEE International Conference on Software Maintenance, ICSM*. 472–481.
[38] R. N. Iyer, S. A. Yun, M. Nagappan, and J. Hoey. Effects of Personality Traits on Pull Request Acceptance. *IEEE Transactions on Software Engineering* (2019), 1–1.
[39] Foutse Khomh and Yann-Gaël Guéhéneuc. Do Design Patterns Impact Software Quality Positively?. In *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*. 274–278.
[40] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*.
[41] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes* 27, 1 (2002), 18–20.
[42] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes* 27, 2 (2002), 20–24.
[43] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 4: questionnaire evaluation. *ACM SIGSOFT Software Engineering Notes* 27, 3 (2002), 20–23.
[44] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes* 27, 5 (2002), 17–20.
[45] Jonathan L. Krein, Lutz Prechelt, Natalia Juristo, Aziz Nanthaamornphong, Jeffrey C. Carver, Sira Vegas, Charles D. Knutson, Kevin D. Seppi, and Dennis L. Eggett. A Multi-Site Joint Replication of a Design Patterns Experiment Using Moderator Variables to Generalize across Contexts. *IEEE Trans. Software Eng.* 42, 4 (2016), 302–321.
[46] Valentina Lenarduzzi, Vili Nikkola, Nyyti Saarimäki, and Davide Taibi. Does Code Quality Affect Pull Request Acceptance? An empirical study. *arXiv preprint arXiv:1908.09321* (2019).

[47] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*. 151–162.

[48] Katsuhisa Maruyama and Kensuke Tokoda. Security-Aware Refactoring Alerting its Impact on Code Vulnerabilities. In *15th Asia-Pacific Software Engineering Conference (APSEC 2008), 3-5 December 2008, Beijing, China*. 445–452.

[49] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. Balancing Agility and Formalism in Software Engineering. Chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, 252–266.

[50] Emerson Murphy-Hill, Chris Parnin, and Andreaw P. Black. How We Refactor, and How We Know It. *Transactions on Software Engineering* 38, 1 (2011), 5–18.

[51] Robert G. Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. *Statistics in Medicine* 17, 8 (1998), 857–872.

[52] Bram Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers.

[53] Matheus Paixao, Anderson Uchoa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. Behind the In-tents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea*. ACM, 11.

[54] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why Developers Refactor Source Code: A Mining-based Study. *ACM Transactions on Software Engineering and Methodology* (2020).

[55] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. An Empirical Investigation of How and Why Developers Rename Identifiers. In *Proceedings of the 2Nd International Workshop on Refactoring (IWoR 2018)*. 26–33.

[56] Shari Lawrence Pfleeger and Barbara A. Kitchenham. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes* 26, 6 (2001), 16–18.

[57] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil A. Ernst, Marco Aurélio Gerosa, Michael W. Godfrey, Michele Lanza, Mario Linares Vásquez, Gail C. Murphy, Laura Moreno, David C. Shepherd, and Edmund Wong. On-demand Developer Documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 479–483.

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

[59] Cagri Sahin, Lori L. Pollock, and James Clause. How do code refactorings affect energy usage?. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*. 36:1–36:10.

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

[61] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Leonardo Murta, and Alexandre Plastino. Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. 1541–1546.

[62] G. Soares, R. Gheyi, and T. Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

[63] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media.

[64] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–Does It Improve Software Quality?. In *Proceedings of the 5th International Workshop on Software Quality (WoSQ '07)*. IEEE Computer Society, Washington, DC, USA, 10–.

[65] Gábor Szoke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 95–104.

[66] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[67] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 483–494.

[68] Jason Tsay, Laura Dabbish, and James D. Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *36th International Conference on Software Engineering, ICSE*

'14, Hyderabad, India - May 31 - June 07, 2014. 356–366.

[69] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 805–816.

[70] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald Gall, and Alberto Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180, 1 (2019), 1–15.

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

[72] Peter Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*. 77–84.