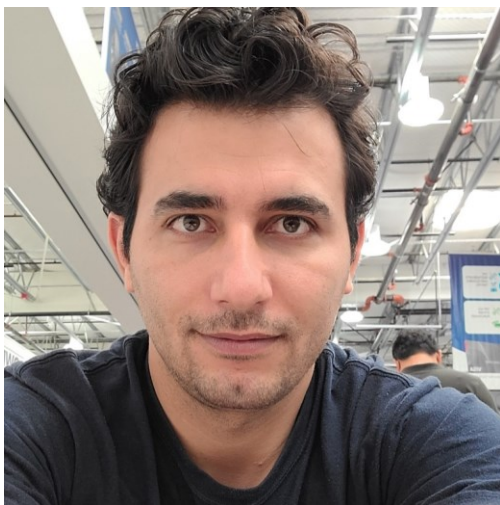


One thousand and one stories: a large-scale survey of software Refactoring



Mohamed Wiem Mkaouer
Department of Software Engineering
Rochester Institute of Technology
mwmvse@rit.edu

Origin of the Study

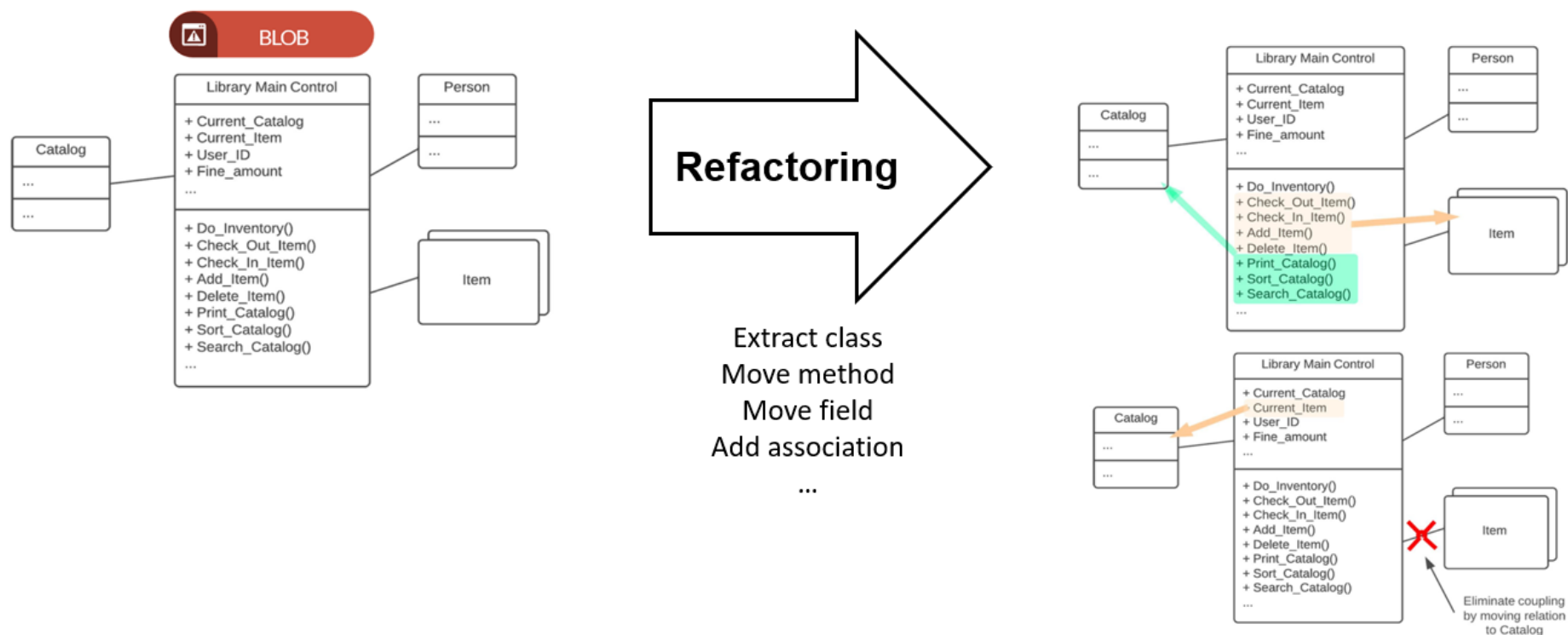


Software Engineer at Xerox

“I noticed that the review of my refactoring changes takes longer than usual to be approved”

Refactoring

- The process of improving a code after it has been written **by changing its internal structure without changing the external behavior.**



Pilot Study at Xerox (Design)



171 Refactoring
Pull Requests



171 non-Refactoring
Pull Requests



Review Duration



exchanged messages

VS

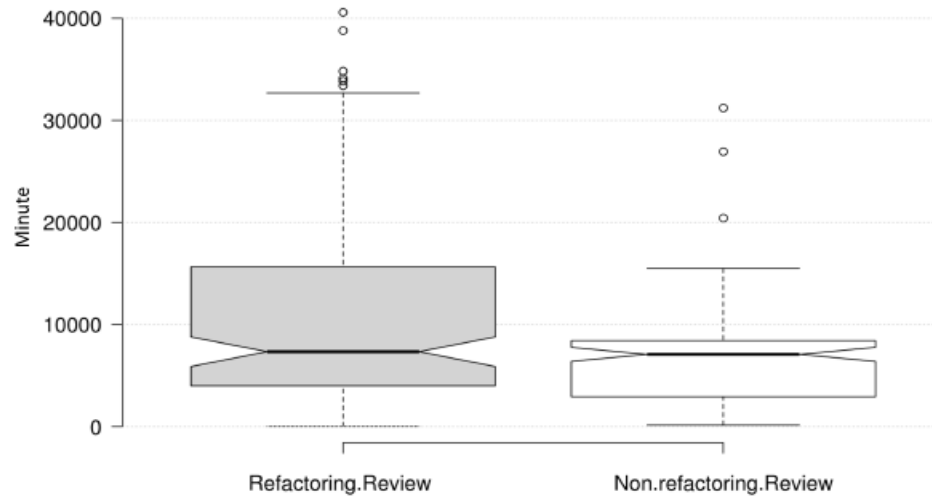


Review Duration

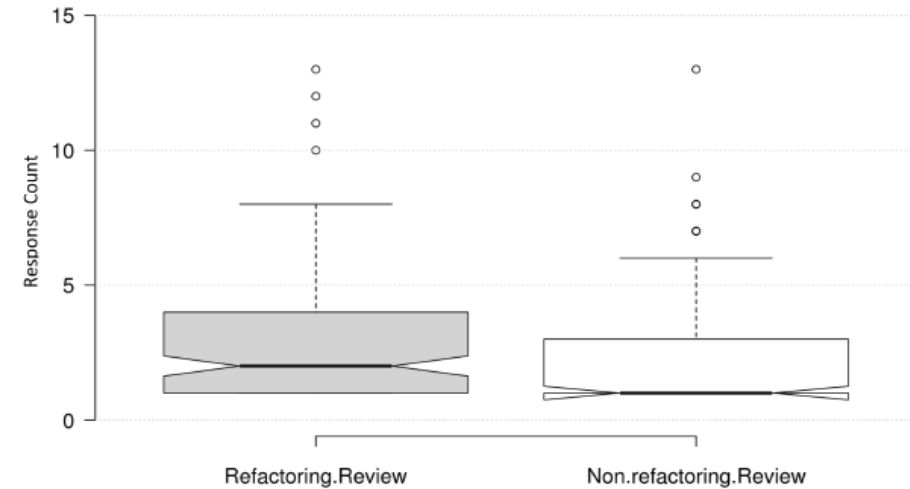


exchanged messages

Challenge 1:



(a) Review duration



(b) Number of exchanged responses

Refactoring code reviews take longer to be completed than the non-refactoring code reviews.

Refactoring code reviews trigger longer discussions between code authors and reviewers before reaching a consensus.

Recommendation 1:



Intent

Understanding the purpose of the intended refactoring

“Removed some dead code”

“Refactored duplicate methods”



Impact

Understanding the impact of the applied refactoring

“[...] to improve readability”

“[...] to reduce complexity”



Instruction

Reporting refactoring operations developers have performed

“Renamed [...]”

“Moved [...]”

“Extracted [...]”



One train may
hide another
one

Recent Refactoring Studies

Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method

Emerson Murphy-Hill and Andrew P. Black

ABSTRACT
Refactoring is the process of changing the structure of existing code without changing its behavior. Refactoring tools, which should make it easier to refactor quickly and correctly. However, many tools do a poor job of communicating the refactoring process and that sometimes refactor slowly, conservatively. This paper we characterize problems with current refactoring tools and demonstrate three new tools to assist a user study that compares these new tools. The results of the study show that satisfaction can be significantly increased by introducing a set of usability recommendations to inspire a new generation of program refactoring tools.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Styles and Standards; D.2.6 [Software Engineering]: Program Analysis
General Terms: Design, Reliability, Human Factors

Keywords: Refactoring, tools, usability, environment

1. INTRODUCTION
Refactoring is the process of changing the structure of existing code without changing the way it behaves. Fowler reports that Extract Method is “a key refactoring” [7]. However, as we will demonstrate, the existence of the Extract Method is fast, error-resistant, and pleasant to use.

Permission to make digital or hard copy personal or classroom use is granted without charge provided that the copies are not made or distributed for profit or commercial advantage, the full copyright notice and title page are reproduced, and the fee code on the title page is reproduced.

Why We Refactor? Confessions of GitHub Contributors

Daniilo Silva
Universidade Federal de Minas Gerais, Brazil
daniilofs@dcc.ufmg.br

ABSTRACT
Refactoring is a widespread practice to improve the maintainability and readability of code. However, there is a limited number of studies investigating the actual motivation for refactoring operations applied by developers. We monitored Java projects hosted on GitHub and identified commonly applied refactorings, and explain the reasons behind their decisions. By applying thematic analysis on the responses, we found evidence that the primary reason for refactoring is to improve the maintainability and readability of code.

CCS Concepts
Software and its engineering; Software development; Software maintenance

Keywords
Refactoring, software evolution, code quality

1. INTRODUCTION
Refactoring is the process of improving the structure of existing code without changing its external behavior. The beginning, the adoption of refactoring is often motivated by the availability of refactoring tools proposed by Fowler [10]. These tools describe the mechanics of each refactoring and demonstrate its application through examples. They also provide a *motivation* for using it, usually associated to the resolution of a problem. For example, EXTRACT METHOD is recommended for large and complex method or to reduce complexity.

Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts

Mehran Mahmoudi
University of Alberta
Edmonton, AB, Canada
mehran@ualberta.ca

Sarah Nadi
University of Alberta
Edmonton, AB, Canada
nadi@ualberta.ca

Nikolaos Tsantalis
Concordia University
Montreal, QC, Canada
tsantalis@cs.concordia.ca

Abstract—With the rise of distributed software development, merge conflicts have become a popular approach that facilitates collaboration between software developers. One of the biggest challenges that developers face when using multiple development branches is dealing with merge conflicts. Conflicts occur when inconsistent changes happen to the code. Resolving these conflicts can be a cumbersome task as it requires prior knowledge about the changes in each of the development branches. A type of change that could potentially lead to complex conflicts is code refactoring. Previous studies have proposed techniques for facilitating conflict resolution in the presence of refactorings. However, the magnitude of the impact that refactorings have on merge conflicts has never been empirically evaluated. In this paper, we perform an empirical study on almost 3,000 well-engineered open-source Java software repositories and investigate the relation between merge conflicts and 15 popular refactoring types. Our results show that refactoring operations are involved in 22% of merge conflicts, which is remarkable taking into account that we investigated a relatively small subset of all possible refactoring types. Furthermore, certain refactoring types, such as EXTRACT METHOD, tend to be more problematic for merge conflicts. Our results also suggest that conflicts that involve refactored code are usually more complex, compared to conflicts with no refactoring changes.

Index Terms—refactoring, git, merge conflict, software evolution

1. INTRODUCTION
Version control systems (VCSs), which keep track of the software development history, have become an essential component of modern software development. With the increase of distributed software development [1], additional coordination tools and processes to facilitate collaboration between team members who may be working on different tasks have been introduced. For example, large software systems commonly make use of branching in distributed version control systems. Developers typically follow a *branch-based development approach*, where new features or bug fixes are developed in separate branches before being integrated into the master branch, or another stable branch [2].

While branching or forking where the developer may have a tracked copy of the work in a separate repository [3] has several advantages such as allowing better separation of concerns and enabling parallel development [4], it still comes at the cost of integration challenges [5]. Once a developer has completed the intended work in a given branch, they need to merge their changes with the rest of the team’s work. At

this point, *merge conflicts* may arise, because of inconsistent changes to the code. Previous studies have shown that up to 16% of merge scenarios lead to conflicts [6]. Developers have to resolve such conflicts before proceeding, which wastes their time and distracts them from their main tasks [7].

There are several types of conflicts and various reasons why a conflict can occur [7]. *Textual conflicts* are those that occur when simultaneous changes occur to the same lines in a file, and are the type of conflicts that popular VCSs such as GIT detect. For example, one developer may have added a new variable declaration `foo` at line 10 of a given file, while the other developer has added another variable declaration `bar` at the same line. When GIT tries to merge both changes, it cannot decide which variable declaration should appear at that line.

Another example of why a conflict can occur is shown in Figure 1. Here, Alice moves function `foo()` from `Foo.java` to `FooHelper.java`, while Bob adds the line `x = 2;` to `foo()` implementation in its original place in `Foo.java`. The figure shows the resulting conflict in `Foo.java`, when Bob tries to merge his code with Alice’s. As shown, the resulting conflict in `Foo.java` shows that the whole function is deleted in one branch, but modified in the other; the number of conflicting lines reported is also large (the size of the whole function `foo`). Given that Bob is not aware that he needs to look at `FooHelper.java` to understand what happened, he would mistakenly think that this is a complex conflict that would take him lots of time to understand and resolve. In reality, the conflict is actually simple: Alice moved the function (a refactoring operation) while Bob added an extra piece of code to it. A simple resolution would be to add the extra piece of code to the new location of the function.

The above example demonstrates how refactorings may complicate the merging process. There have been a few studies that investigated how to deal with refactorings during merging. For example, Dig et al. [8] previously argued that since refactorings cut across module boundaries and affect many parts of the system, they make it harder for VCSs to merge the changed code. They proposed *refactoring-aware merging*, with the argument that if a merging tool understands the refactorings that took place, it may be able to automatically resolve the conflict and save the developer’s time. In the

Alternate Refactoring Paths Reveal Usability Problems

Mohsen Vakilian and Ralph E. Johnson

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Use, Disuse, and Misuse of Automated Refactorings

The Usability (or Not) of Refactoring Tools

Anna Maria Eilertsen
Department of Informatics
University of Bergen
Bergen, Norway
anna.eilertsen@iuh.no

Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, Canada
murphy@cs.ubc.ca

Abstract—Although software developers typically have access to numerous refactoring tools, most developers avoid using these tools despite their benefits. Researchers have identified many reasons for the disuse of refactoring tools, including a lack of awareness by the developers, a lack of predictability of the tools, and a lack of need for the tools. In this paper, we build on this earlier work and employ the ISO 9241-11 definition of usability to develop a theory of usability for refactoring tools. We investigate existing refactoring tools using this theory by analyzing how 17 developers experience refactoring tools in three software change tasks we asked them to perform. We analyze qualitatively the resulting interview transcripts based on our theory and report on a number of observations that can inform tool designers interested in improving the usability of refactoring tools. For instance, we found a desire for developers to guide how a refactoring tool changes the code and a need for refactoring tools to describe changes made to developers. Refactoring tools are currently expected to preserve program behavior. These observations indicate that it may be necessary to give developers more control over this property, including the ability to relax it, for the tools to be usable; that is, for the tools to be effective, efficient and satisfying for the developer to employ.

Keywords—automation, tool usability, software evolution

1. INTRODUCTION
During software evolution and maintenance, developers frequently apply *refactoring* operations to source code [1], [2] for the purpose of improving its quality [3] or preparing and completing functional code changes [4], [5]. *Refactoring tools* automate the application of refactoring operations and reduce development costs by automatically performing code changes that are time-consuming and error-prone for developers to apply manually.

As a simple example, a developer may change the name of a method in order to better represent its functionality. If the program should behave as before, all references to the method throughout the program must be updated to use the new name. This *rename* refactoring operation, and many others, such as extracting, moving and inlining program elements, occur during software change activities [1]–[4], [6] and enjoy extensive automated support in most mainstream development environments, such as Eclipse [7] and IntelliJ [8].

Developers’ refactoring activities have been studied for over three decades and much effort has been made to automate refactoring support [9]. While some automated operations, like

This work is supported by the Research Council of Norway under grant number 250683 (Co-Evo).

the aforementioned *rename*, have been readily embraced by software developers, most refactoring operations are applied manually [1], [4], [10], [11] and refactoring tools are, in fact, *disused* [2].

Empirical studies indicate that developers disuse refactoring tools that are present in their programming environments—even when they are aware of them—due to usability issues [1], [2], [4], [10], [12]. Researchers have proposed various approaches to the reported usability problems [13]–[20], most of which are aimed at improving singular usability aspects such as speed [21], selecting code [12] or choosing the right refactoring operation [18]–[20]. There is no comprehensive approach to refactoring tool usability nor do we understand what causes a developer to find a refactoring tool trustworthy or predictable [22], [23].

In this paper, we seek to provide a framework to help guide refactoring tool designers in creating refactoring tools that developers choose to use. We derive a theory of the usability of refactoring tools (Section III) based on the ISO 9241-11 definition of usability [24]. This new theory differs from existing theories (Section II) in two ways. First, it considers a more common context of use of the tools, namely to prepare or complete functional changes to a system. Second, it considers what developers seek as an experience in using the tools.

We then use the newly proposed theory to study the use of refactoring tools by 17 software developers attempting three functional software change tasks on a non-trivial software system. As the participating software developers performed the tasks, we captured their think-aloud [25] descriptions. We also conducted semi-structured interviews with the participants after each task and at the end of a study session.

We analyze the transcripts captured from the 32 hours of study sessions by employing the introduced theory (Section III) and findings from previous empirical studies of refactoring tools (Section II). We use a coding approach to investigate the participants’ comments according to five usability factors—*effectiveness*, *efficiency*, *satisfaction*, *predictability* and *trust*, finding that the participants’ experience with refactoring tools is dominated by their perceptions of the predictability of the tools. Next, we perform card sorts [26] of the comments associated with each of these factors: i.e., we organize the comments into groups until meaningful categories emerge. A synthesis of the results indicates a need for refactoring tools to better communicate their capabilities

and the perceived utility of the growing interest in automated refactorings [17]—has not received enough attention. Refactoring tools with usability issues and interviews [24] find real usability number of participants own to be suitable for problems, e.g., those that study or the ones that an interview. This is not the tool only for of tasks, and a small number of participants. Certain events and injuries. Reflect our future strategies. How would they have been affected? Critical thinking of both quantitative and qualitative data. We studied 26 developers on their code for a total of three months, and collected their automated refactorings. In our quantitative data participants to take a more behavioral data. Then, we used human-automation interface, and misuse of automated refactorings referred refactorings to perform manually. *Disuse* of refactorings to perform manually. *Disuse* of refactorings to perform manually. *Disuse* of refactorings to perform manually.

How users interact with the tool. We have found that a method of invoking a certain number of refactoring operations, we have found several utilization of automated refactoring, naming, trust, presence (Section IV). Third, we will continue an automated error or warning. This in property of automated refactoring. In addition, we uses of the refactoring tool proposed alternative ways seen on the findings of our G, and V-C).

Recent Refactoring Studies

Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method

Emerson Murphy III and Andrew P. Black
Rochester State University, P.O. Box 750

Why We Refactor? Confessions of GitHub Contributors

Emerson Murphy III

Andrew P. Black

ABSTRACT

Refactoring is the process of changing the internal structure of existing code without changing its external behavior. It is a fundamental technique for maintaining and improving the quality of software. This paper presents the results of a study of GitHub contributors' reasons for refactoring. The study found that the most common reasons for refactoring were to improve code readability, to reduce code duplication, and to improve code maintainability. The study also found that contributors often used automated refactoring tools, but that they often used these tools in a way that was not intended by the tool designers. This suggests that there is a need for better automated refactoring tools that are more user-friendly and that can better understand the context of the code being refactored.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.4 [Software Engineering]: Testing

General Terms: Design, Reliability, Experimentation

Keywords: Refactoring, usability, developer preferences, automated refactoring tools

1. INTRODUCTION

Refactoring is the process of changing the internal structure of existing code without changing its external behavior. It is a fundamental technique for maintaining and improving the quality of software. This paper presents the results of a study of GitHub contributors' reasons for refactoring. The study found that the most common reasons for refactoring were to improve code readability, to reduce code duplication, and to improve code maintainability. The study also found that contributors often used automated refactoring tools, but that they often used these tools in a way that was not intended by the tool designers. This suggests that there is a need for better automated refactoring tools that are more user-friendly and that can better understand the context of the code being refactored.

Alternate Refactoring Paths Reveal Usability Problems

Matthew Jettison and David S. Johnson
University of Illinois at Chicago, Chicago, IL

Use, Abuse, and Misuse of Automated Refactoring

William Walker, Nicholas Chen, Ben Hayes, Rajarathnam Subramanian, Brian F. Keller, Raghav K. Arora
University of Illinois at Chicago, Chicago, IL

The Usability (or Not) of Refactoring Tools

Carl C. Healy
University of Illinois at Chicago, Chicago, IL

- Refactoring tools are underused
- Developer preference of manual refactoring
- Automated refactoring: lack of trust
- No formal refactoring documentation

Lack of refactoring culture?

JetBrains IDEA

Lack of trust (10 instances) was the most frequent reason. Some developers do not trust refactoring tools for complex operations that involve code manipulation and only use them for renaming or moving:

"I don't trust the IDE for things like this, and usually lose other comments, notation, spacing from adjacent areas."

"I'd say developers are reluctant to let a tool perform anything but trivial refactorings, such as the ones you picked up on my commit."

On the other hand, some developers also think that tool support is unnecessary in simple cases (8 instances). Sometimes the operation may involve only local changes and is trivial to do by hand. Thus, calling a special operation to do it is considered unnecessary, as illustrated by this comment: *"Automated refactoring is overkill for moving some private fields."*

Additionally, developers also mentioned: lack of tool support for the specific refactoring they were doing (6 instances), not being familiar with refactoring features of the IDE (3 instances), and not realizing they could use refactoring tools at the moment of the refactoring (2 instances).

5.3 What IDEs developers use for refactoring?

When answering to our emails, 83 developers spontaneously mentioned which IDE they use. Therefore, we decided to investigate these answers, specially because our study is not dependent on any IDE, and thus differs from previous studies which are usually based only on Eclipse data [23, 24]. Table 8 shows the most common IDEs mentioned in these answers and the percentage of refactorings performed automatically in these cases. 139 developers (63%) did not explicitly mention an IDE when answering this question.

we did not find any instances of the themes *encapsulate field* and *hide message chain*, reported in [36], which are related to code smell resolution. We assume these different themes are due to the nature of the examined projects, since [36] examined only three libraries and frameworks, while in this study we examined 124 projects from various domains including standalone applications. By comparing to the code symptoms that initiate refactoring reported in the study by Kim et al. [17], we found the *readability, reuse, testability, duplication*, and *dependency* motivation themes in common.

Most of the refactoring motivations we found have the intention to facilitate or even enable the completion of the maintenance task that the developer is working on. For instance, *extract reusable method, introduce alternative method signature*, and *facilitate extension* are among the most frequent motivations, and all of them involve enhancing the functionality of the system. Therefore, EXTRACT METHOD is a key operation to complete other maintenance tasks, such as adding a feature or fixing a bug. In contrast, only two out of the 11 motivations we found (*decompose method to improve readability and remove duplication*) are targeting code smells. This finding could motivate researchers and tool builders to design refactoring recommendation systems [35, 30, 33, 12, 18, 37] that do not focus solely on detecting refactoring opportunities for the sake of code smell resolution, but can support other refactoring motivations as well.

We also observe that developers are seriously concerned about avoiding code duplication, when working on a given maintenance task. They often use refactorings—especially EXTRACT METHOD—to achieve this goal, as illustrated by the following comments:

"I need to add a check to both the then- and the else-part of an if-statement. This resulted in more duplicated code than

bug fix required implication. That up the replicated

ically performed, the most common ATTRIBUTE, and ats, so that they evance.

In a field study that most refacour study, in IDEs, we found rmed. However, nd to use more E users. More-METHOD refac-% in our study ntages of manu-we should keep e refactorings, like th tool support. : al. [23], where rmed manually

(considering also renamings), we detected significantly more automated refactorings. We suspect this difference may be due to two reasons. First, automated refactoring tools may have become more popular and reliable over the last years.

Table 8: IDE popularity

IDE	Occurrences	Automated %
Editor not mentioned	139	12%
IntelliJ IDEA	51	71%
Eclipse	18	44%
NetBeans	8	50%
Android Studio	4	25%
Text Editor	2	0%

TRACT METHOD IS THE "SWISS ARMY KNIFE OF REFACTORINGS" [36]. It is the refactoring with the most motivations (11 in total). In comparison to [36], there is an overlap in the reported motivation themes for EXTRACT METHOD. We found some new themes, such as *improve testability and enable recursion*, but

Why We Refactor? Confessions of GitHub Contributors

Danilo Silva
Universidade Federal de
Minas Gerais, Brazil
danilofs@dcc.ufmg.br

Nikolaos Tsantalis
Concordia University
Montreal, Canada
tsantalis@cse.concordia.ca

Marco Tulio Valente
Universidade Federal de
Minas Gerais, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Refactoring is a widespread practice that helps developers to improve the maintainability and readability of their code. However, there is a limited number of studies empirically investigating the actual motivations behind specific refactoring operations applied by developers. To fill this gap, we monitored Java projects hosted on GitHub to detect recently applied refactorings, and asked the developers to explain the reasons behind their decision to refactor the code. By applying thematic analysis on the collected responses, we compiled a catalogue of 44 distinct motivations for 12 well-known refactoring types. We found that refactoring activity is mainly driven by changes in the requirements and much less by code smells. EXTRACT METHOD is the most versatile refactoring operation serving 11 different purposes. Finally, we found evidence that the IDE used by the developers affects the adoption of automated refactoring tools.

CCS Concepts

•Software and its engineering → Software evolution; Maintaining software; Software maintenance tools;

Keywords

Refactoring, software evolution, code smells, GitHub

1. INTRODUCTION

Refactoring is the process of improving the design of an existing code base, without changing its behavior [27]. Since the beginning, the adoption of refactoring practices was fostered by the availability of refactoring catalogues, as the one proposed by Fowler [10]. These catalogues define a name and describe the mechanics of each refactoring, as well as demonstrate its application through some code examples. They also provide a *motivation* for the refactoring, which is usually associated to the resolution of a code smell. For example, EXTRACT METHOD is recommended to decompose a large and complex method or to eliminate code duplication.

As a second example, MOVE METHOD is associated to smells like Feature Envy and Shotgun Surgery [10].

There is a limited number of studies investigating the real motivations driving the refactoring practice based on interviews and feedback from actual developers. Kim et al. [17] explicitly asked developers "in which situations do you perform refactorings?" and recorded 10 code symptoms that motivate developers to initiate refactoring. Wang [40] interviewed professional software developers about the major factors that motivate their refactoring activities and recorded human and social factors affecting the refactoring practice. However, both studies were based on general-purpose surveys or interviews that were not focusing the discussion on specific refactoring operations applied by the developers, but rather on general opinions about the practice of refactoring. **Contribution:** To the best of our knowledge, this is the first study investigating the motivations behind refactoring based on the actual explanations of developers on specific refactorings they have recently applied. To fill this gap on the empirical research in this area, we report a large scale study centered on 463 refactorings identified in 222 commits from 124 popular, Java-based projects hosted on GitHub. In this study, we asked the developers who actually performed these refactorings to explain the reasons behind their decision to refactor the code. Next, by applying thematic analysis [6], we categorized their responses into different themes of motivations. Another contribution of this study is that we make publicly available¹ the data collected and the tools used to enable the replication of our findings and facilitate future research on refactoring.

Relevance to existing research: The results of this empirical study are important for two main reasons.

First, having a list of motivations driving the application of refactorings can help researchers and practitioners to infer rules for the automatic detection of these motivations when analyzing the commit history of a project. Recent research has devised techniques to help in understanding better the practices of code evolution by identifying frequent code change patterns from a fine-grained sequence of code changes [26], isolating non-essential changes in commits [13], and untangling commits with bundled changes (e.g., bug fix and refactoring) [7]. In addition, we have empirical evidence that developers tend to interleave refactoring with other types of programming activity [24], i.e., developers tend to *flask refactor*. Therefore, knowing the motivation behind a refactoring can help us to understand better other related changes in a commit. In fact, in this study we found

¹<http://aseg-ufmg.github.io/why-we-refactor>

Research Goal

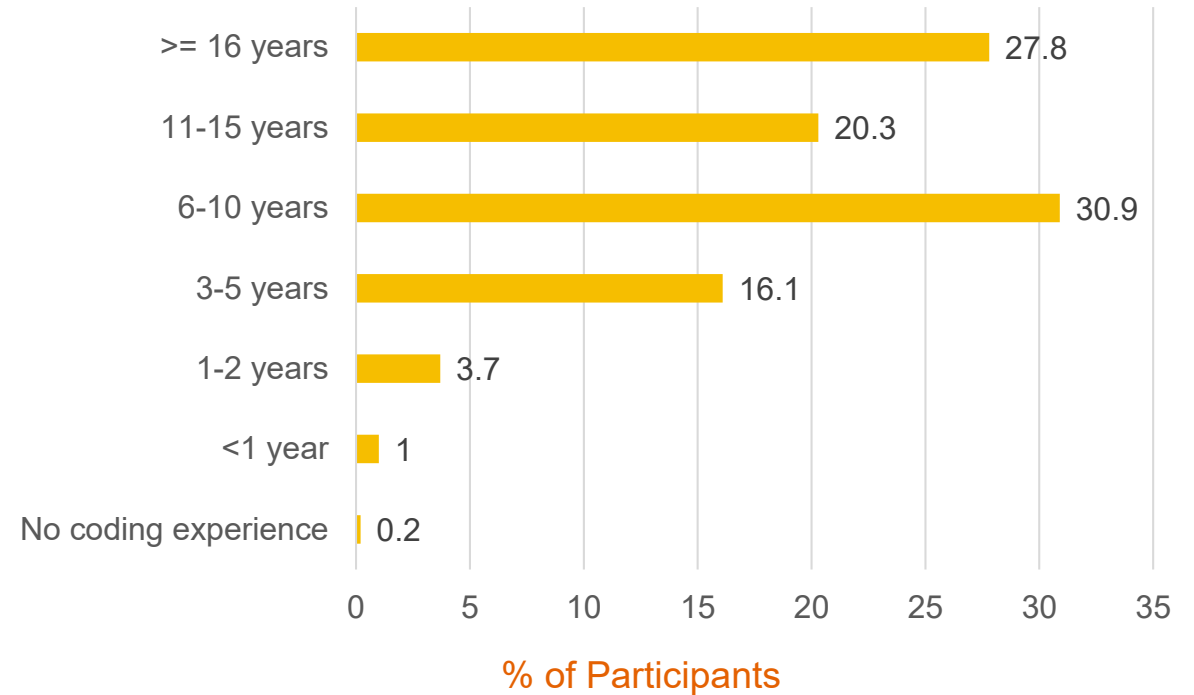
How do developers use IntelliJ to refactor code?

Study Design



1183 Developers

Participants professional development experience



How Code Gets Refactored in IntelliJ?

How Code Gets Refactored in IntelliJ?

	Used IDE refactoring	Used Find and Replace	Used Copy and Paste	Edited manually	Didn't have this scenario
Renaming a class, method, variable, or symbol	85.8%	46.2%	21.3%	29.8%	0.3%
Extracting a method or a variable from existing code	54.7%	20.7%	30.4%	33.2%	10.7%
Moving code to another file	38.6%	12.2%	57.5%	30.8%	4.4%

Rename



Extract



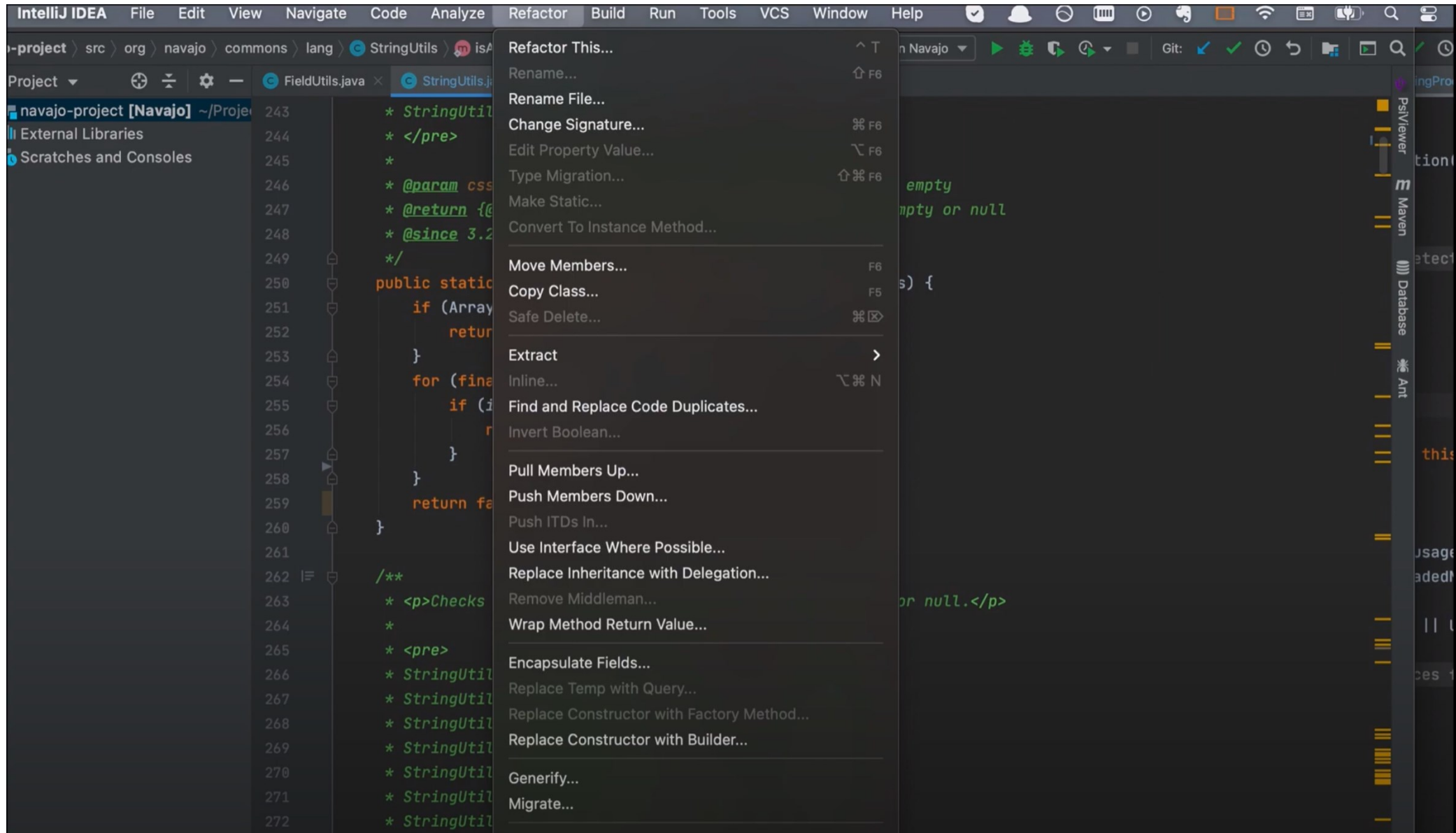
Move



Challenge 2:



Recommendation 2:



Questions?

Contact me:
mwmvse@rit.edu

Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox

Eman Abdullah AlOmar^{*}, Hussein AlRubaye[†], Mohamed Wiem Mkaouer^{*}, Ali Ouni[‡], Marouane Kessentini[§]

^{*}Rochester Institute of Technology, Rochester, NY, USA

[†]Xerox Corporation, Rochester, NY, USA

[‡]ETS Montreal, University of Quebec, Montreal, QC, Canada

[§]University of Michigan, Dearborn, MI, USA

eman.alomar@mail.rit.edu, hussein.alrubaye@xerox.com, mwmvse@rit.edu, ali.ouni@etsmtl.ca, marouane@umich.edu

Abstract—Modern code review is a common and essential practice employed in both industrial and open-source projects to improve software quality, share knowledge, and ensure conformance with coding standards. During code review, developers may inspect and discuss various changes including refactoring activities before merging code changes in the code base. To date, code review has been extensively studied to explore its general challenges, best practices and outcomes, and socio-technical aspects. However, little is known about how refactoring activities are being reviewed, perceived, and practiced.

This study aims to reveal insights into how reviewers develop a decision about accepting or rejecting a submitted refactoring request, and what makes such review challenging. We present an industrial case study with 24 professional developers at Xerox. Particularly, we study the motivations, documentation practices, challenges, verification, and implications of refactoring activities during code review.

definition, is not intended to alter the system's behavior, but to improve its structure, so its review may differ from other code changes. Yet, there is not much research investigating how developers review code refactoring. The research on refactoring has been focused on its automation by identifying refactoring opportunities in the source code, and recommending the adequate refactoring operations to perform [6]–[8]. Moreover, the research on code reviews has been focused on automating it by recommending the most appropriate reviewer for a given code change [3]. However, despite the critical role of refactoring and code review, the innate relationship between them is still largely unexplored in practice.

The goal of this paper is to understand how developers review code refactoring, *i.e.*, what criteria developers rely on to develop a decision about accepting or rejecting a submitted

One Thousand and One Stories: A Large-Scale Survey of Software Refactoring

Yaroslav Golubev
 JetBrains Research
 Saint Petersburg, Russia
 yaroslav.golubev@jetbrains.com

Zarina Kurbatova
 JetBrains Research
 Saint Petersburg, Russia
 zarina.kurbatova@jetbrains.com

Eman Abdullah AlOmar
 Rochester Institute of Technology
 Rochester, United States
 eman.alomar@mail.rit.edu

Timofey Bryksin
 JetBrains Research
 Higher School of Economics
 Saint Petersburg, Russia
 timofey.bryksin@jetbrains.com

Mohamed Wiem Mkaouer
 Rochester Institute of Technology
 Rochester, United States
 mwmvse@rit.edu

ABSTRACT

Despite the availability of refactoring as a feature in popular IDEs, recent studies revealed that developers are reluctant to use them, and still prefer the manual refactoring of their code. At JetBrains, our goal is to fully support refactoring features in IntelliJ-based IDEs and improve their adoption in practice. Therefore, we start by raising the following main questions. How exactly do people refactor code? What refactorings are the most popular? Why do some developers tend not to use convenient refactoring tools provided by modern IDEs?

Stories: A Large-Scale Survey of Software Refactoring. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Refactoring [12] is traditionally defined as the process of improving the internal code structure without altering its external behavior. Since this practice had been introduced to a wide audience of software engineers, it has become a crucial tool to maintain high-quality software and to reduce its technical debt. Several refactoring types,

AlOmar, et al. "Refactoring practices in the context of modern code review: An industrial case study at Xerox." In *ICSE-SEIP*, pp. 348-357. 2021.

Golubev et al. "One thousand and one stories: a large-scale survey of software refactoring." In *FSE*, pp. 1303-1313. 2021.