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?

In this paper, we investigate the raised questions through the design and implementation of a survey targeting 1,183 users of IntelliJ-based IDEs. Our quantitative and qualitative analysis of the survey results shows that almost two-thirds of developers spend more than one hour in a single session refactoring their code; that refactoring types vary greatly in popularity; and that a lot of developers would like to know more about IDE refactoring features but lack the means to do so. These results serve us internally to support the next generation of refactoring features, as well as can help our research community to establish new directions in the refactoring usability research.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Maintaining software**.

KEYWORDS

Refactorings, IDE Refactoring Features, Software Maintenance

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, involving renaming, moving, and extracting elements have been implemented as actionable tools in modern Integrated Development Environments (IDEs), providing developers with an automatic and safe way to apply these predefined code transformations [32, 33, 36].

Even though all modern IDEs usually have a top-level menu with various options devoted to refactoring, several recent surveys report that developers are often reluctant to adopt these features and still manually refactor their code [17, 35]. Despite the high-level maturity of IDEs and the safety and interactivity of their tools, manual refactoring is still widely adopted regardless of its drawbacks and error-proneness.

While several studies have recently highlighted the lack of automated refactoring usage [1, 15, 17, 30, 35, 40], little is known regarding the reasons hindering the widespread adoption of automated refactoring, especially when it is offered as a built-in feature in modern IDEs. Furthermore, existing studies were limited by investigating only a few types of refactoring, as well as by the number of developers surveyed.

The goal of this paper is to share the results of a large-scale survey about refactoring conducted by JetBrains Research, to reflect on what refactorings developers actually use, as well as to raise the awareness of the current usability challenges that developers face when they use the IDE to refactor their code. In particular, we designed a survey of 20 questions to investigate several dimensions related to (1) general background information about respondents, (2) developers' familiarity with refactoring in general, (3) how developers tend to refactor their code, (4) the extent to which they are familiar with the IDE built-in refactoring functionalities, (5) along with their degree of adoption and thoughts on specific refactoring features of IDEs, namely the UNDO and PREVIEW. Our survey was sent to paid subscribers of the IntelliJ Platform-based IDEs

(IntelliJ IDEA, PyCharm, WebStorm, and others), and we received 1,183 complete responses, achieving a response rate of 6.0% and a completion rate of 88.9%.

We provide the refactoring community with a variety of insights that are currently being investigated by JetBrains IDE development teams to plan future enhancements of the refactoring tools. The raw data of the survey is available online.[1] Notably, the survey has revealed the following:

- While refactoring is a regular and recurrent practice supporting the software development cycle, two-thirds of respondents confirmed that their refactoring sessions can take up to an hour or even longer.
- Some refactorings, such as *Rename* and *Extract* entities, are more popular and intuitive than others, such as *Pull Up* and *Push Down* entities. Despite the existence of built-in refactoring tools, some developers opt for refactoring their code using manual practices: for instance, *Find and Replace* is popular for *Renaming*, *Copy and Paste* is typically used for *Moving*.
- The familiarity of developers with IDE refactoring features varies from one refactoring type to another. Also, the adoption of an IDE feature correlates with the popularity of the refactoring itself. For instance, *Rename* refactoring has been found to be the most popular one, and its built-in RENAME tool is also heavily used. This is in contrast with *Pull Up / Push Down* refactorings, which are the least popular, and consequently their corresponding tools are used significantly less frequently.
- The possibility of PREVIEW and UNDO helps in consolidating the usage of refactoring tools, as developers tend to trust code transformations more and understand their impact better. Nevertheless, these functionalities can be considered too complex when developers are performing what they believe to be simple and intuitive refactorings.

2 RELATED WORK

Some refactoring techniques and formalisms to guarantee program preservation have been reported in an extensive survey study by Mens and Tourwe [20]. The authors discussed the existing literature in terms of refactoring activities and their automation techniques. They reported different types of software artifacts being refactored, along with existing refactoring tool support, and the impact of refactoring in the software process. Murphy-Hill & Black [22] surveyed 112 Agile Open Northwest conference attendees and found that refactoring tools are underused by professional programmers. At Microsoft, Kim et al. [17] surveyed 328 professional software engineers to investigate when and how they do refactoring. When surveyed, the developers mentioned the main benefits of refactoring to be: improved readability (43%), improved maintainability (30%), improved extensibility (27%), and fewer bugs (27%). When asked what provokes them to refactor, the main reason provided was poor readability (22%). Only one code smell (*i.e.*, code duplication) was mentioned (13%). Sharma et al. [34] surveyed 39 software architects to ask about the problems they are facing whenever they refactor their systems and the limitations of existing refactoring

tools they use. Their main findings are: (1) fear of breaking the code restricts developers from adopting refactoring techniques, (2) lack of awareness of the impact of refactoring on code quality is a major obstacle to refactoring tasks, and (3) developers feel reluctant to adopt refactoring because it might result in wasting their resources. Oliveira et al. [25] surveyed 107 developers about their refactoring output, using 7 refactoring types applied to pilot software systems. They found significant differences in the outputs for the same tasks due to the differences in the IDE refactorings they used. In their extended work [26], Oliveira et al. confirmed that refactoring implementations of various IDEs (Eclipse, NetBeans, etc.) have differences in all refactoring types. They also reported that these IDEs have different input parameters to apply refactorings.

Our study complements the ongoing effort of previous studies by providing more in-depth insights regarding the challenges faced by developers, specifically when using modern IDEs. Our study is also the first to attract 1,183 developers, becoming the largest refactoring survey in literature.

3 STUDY DESIGN

3.1 Pilot Survey

Since this survey was going to be distributed to a wide range of participants, it was critical to ensure that our questions properly convey the points we are seeking answers to. Therefore, we performed a pilot study for the purpose of refining our questions and survey protocol. The pilot version of the survey contained the following questions: (1) What is your experience in software engineering and what programming languages do you regularly use? (2) Provided a list of popular refactorings, please select whether you know each one of them in general and as an IDE feature. (3) For the same refactorings, please select which ones you use and how often. (4) What are your general thoughts about the current state of automatic refactorings? (5) What are your negative experiences with automatic refactorings? (6) If you had cases when you wanted to perform a refactoring, but decided not to, what were the reasons? (7) How often do you use the PREVIEW IDE feature when performing a refactoring? What are your main reasons for using it? (8) How often do you use the UNDO action after applying a refactoring? What are your main reasons for using it?

The pilot version of the survey was reviewed by the members of the Market Research and Analytics team at JetBrains, who are experienced with survey design and execution. We have received the following feedback to refine the questions.

- In questions (2) and (3), we immediately show a list of refactoring names to respondents, thus possibly alienating developers who may perform refactorings but do not know them by their names. Instead, we should tune our questions according to the participants' familiarity with refactorings.
- It is better to strictly divide questions about refactoring activities in general and questions about IDE refactoring features in order not to confuse less experienced participants.
- To avoid generic responses, it is recommended to tie respondents to a specific time frame: asking about their refactoring experience during the past month, during one programming session, etc.

---

[1]Raw data of the survey: https://zenodo.org/record/4923175

Table 1: The summary of survey questions. Grey numbers near certain questions indicate that the presence of this question conditionally depends on another question, specified by the number

	I. Background
Question 1	How many years of coding experience do you have?
Question 2	What programming languages do you regularly use?

	II. Familiarity with refactorings
Question 3	In the past month, how often have you performed any code refactoring?
Question 4 (3)	Have you renamed anything in your code or project structure in the past month?
Question 5 (3)	Have you moved any code from one location in the project to another?
Question 6 (3)	During this time, did you ever refactor code for an hour or more in a single session?

	III. Refactoring approaches
Question 7	For the following scenarios, please select all the approaches you have used in the past month. (*Renaming a class, method, variable, or symbol / Extracting a method or a variable from existing code / Moving code to another file*)
Question 8 (7)	In these scenarios, what were your main reasons for not using the IDE refactoring feature?
Question 9	For the following scenarios, please select all the approaches you have used in the past month. (*Inlining a variable or method / Changing the signature of an existing function / Moving a method up or down the class hierarchy*)
Question 10 (9)	In these scenarios, what were your main reasons for not using the IDE refactoring feature?

	IV. IDE refactoring features
Question 11	How familiar are you with the following IDE refactoring features? (Rename file, class, method, symbol, etc. / Extract method, variable, component, etc. / Move / Inline variable or method / Change signature / Pull Up or Push Down member)
Question 12 (11)	Please think about the last several times you used IDE refactorings. How happy were you with the overall experience?
Question 13 (12)	Please tell us a bit more about your experience.
Question 14 (11)	How often do you undo or revert an IDE refactoring action because you're unhappy with the result?
Question 15 (14)	The last few times you undid or reverted an IDE refactoring feature, what were the reasons?

	V. Previewing refactorings
Question 16	When refactoring the code, do you find it useful to preview all of your changes before applying them?
Question 17 (16)	For what types of changes do you find this feature most useful?
Question 18 (16)	What are your main reasons for wanting to see a refactoring preview?
Question 19 (16)	What are your main reasons for not finding a preview useful?

	VI. Final thoughts
Question 20	Please share any thoughts or feedback you have about using the IDE refactorings.

- Question (6) is too broad, it is better to split it into more specific questions. Question (7) is restricted to participants who *use* previews, whereas it would be also interesting to learn the reasons for not using that feature.

This feedback allowed us to reformulate some questions and consolidate others. The resulting survey was then approved by the Market Research and Analytics team, and its questions are enumerated in Table 1.

3.2 Final Survey

Now, let us describe all the research dimensions encapsulated in the final version of the survey.

The complete list of questions is presented in Table 1. There was a total of 20 questions in the survey: 9 single choice, 8 multiple choice (in 4 of which a write-in *Other* answer was available), and 3 open-ended. Also, since our survey targeted different groups of developers, it contained *conditional questions* and one case of *branching*. Conditional questions allow us to gather deeper insights and target specific groups of respondents, and branching allows tailoring questions to a specific group in case groups do not intersect (for example, when some developers like a feature, and the others do not). According to the survey methodology and the guidelines proposed by Kitchenham and Pfleeger [10, 14, 18], conditional questions and branching ensure that respondents are asked only those questions that apply to them while allowing us to gather deeper insights. You can find more information about conditional questions and branching in our supplementary data package [13]. Let us now briefly overview the main sections of the survey.

I **Background**. The first introductory section had just two basic questions related to the respondent's coding experience and the programming languages they use.

II **Familiarity with refactorings**. The second section aimed to determine the respondent's familiarity with refactorings. We asked them how often they have been recently refactoring code. In case the respondents said that they do it rarely or never at all, we followed up by asking how often they have *renamed* or *moved* code elements to ensure that their previous selection was not due to their unfamiliarity with the term *refactoring*. Finally, we asked the respondents whether they have spent one hour or more in a single session refactoring their code.

III **Refactoring approaches**. The third section was dedicated to discovering whether developers actually carry out refactorings with IDE refactoring tools or manually. In the cases when they did not refactor using IDE tools, we asked a follow-up question to inquire about the reasons for that.

IV **IDE refactoring features**. In the fourth section, we concentrated on the automatic refactoring tools of IDEs. We asked the respondents how often they use each type of refactoring features (*e.g.*, Rename). If at least one refactoring feature was regularly used, we presented more questions to gauge their satisfaction with these features. We also inquired about how often respondents had to undo an applied refactoring.

V **Previewing refactorings**. The fifth section discussed the usage of previews when refactoring code. We divided respondents into those who find previews useful and those who do not, and then dived deeper into the reasons for both of these views.

VI **Final thoughts**. Finally, in the last section of the survey, we gave the respondents the opportunity to share any refactoring-related feedback.

3.3 Surveying Process

We selected respondents from the list of paid subscribers of IntelliJ-based IDEs (IntelliJ IDEA, PyCharm, CLion, WebStorm, etc.) [31] who had previously agreed to receive invitations to such surveys by email. In total, the mailing list consisted of 20,000 developers, compiled evenly from users of different IntelliJ-based IDEs. This allowed us to mitigate possible skews towards a given IDE or specific languages and inquire about broader aspects of refactoring.

19,860 emails were successfully delivered, respondents were given three weeks to fill the survey. In the end, we received 1,330 responses, out of which 1,183 were complete. This corresponds to a **response rate** of 6.0% and a **completion rate** of 88.9%. We considered the obtained response rate to be satisfactory, as it is close to other studies in the field that had reported a response rate between 5.7% [29] and 7.9% [19]

The first two questions in the survey gave us some background information on the developers. The question about coding experience (**Question 1**) demonstrated that 27.8% of the participants have more than 16 years of experience, a total of 48.1% have more than 10 years of experience, and as much as 79% have more than 5 years of experience. Only 2 respondents (0.2%) in our survey answered *I do not have any coding experience*, and were therefore taken to the end of the survey.

Question 2 allowed our participants to select up to 3 languages that they are familiar with. The most popular languages by the percentage of respondents in our survey are: JavaScript (46.6%), Python (28.3%), PHP (27.2%), Java (24.4%), and SQL (21.4%). These languages constitute some of the most popular programming languages used in different software engineering domains [11]. Among other languages selected by the participants are Typescript, C/C++, C#, Go, Ruby, and others.

Overall, the results of this survey represent the opinions of more than a thousand developers with a high average experience in different popular programming languages, which can serve as a practical window to the current general state of refactoring usage.

4 RESULTS

In this section, we describe the results of the survey, divided into five parts, corresponding to sections **II–VI** mentioned in Section 3.2 and shown in Table 1.
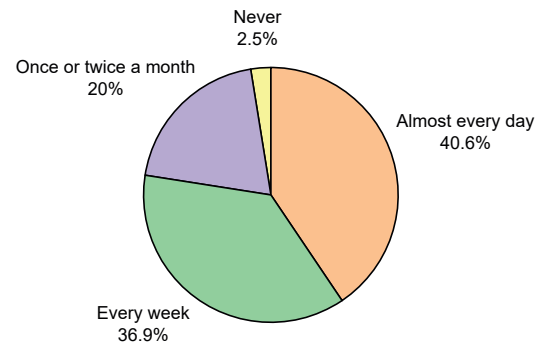
Because of the conditional questions and branching, questions in our survey have a different number of respondents answering them. To avoid creating confusion with numbers not summing up, we decided to present the results of all questions in the form of percentages with respect to the number of respondents who answered them. For all survey questions, the number of respondents is explicitly stated in captions of the figures and tables. Also, to avoid confusion in similar terms, we mark *refactorings in general* in cursive and mark IDE REFACTORING FEATURES in small caps.

To process the open-ended questions, we used the open coding technique based on guidelines provided by Cruzes et al. [9]. For each question, firstly, the list of possible categories of answers was

compiled. This was done independently by the first two authors in two iterations. On the first iteration, each author studied the responses and gave them possible labels, and on the second iteration, each author reduced the overlap between the labels and drew up the final list of categories. After this, the first two authors compared their lists of obtained categories and compiled the final list together. Finally, using this final list, they independently did a third iteration and labeled each response (no category, one specific category, or several categories in the case of long responses). After this, they compared and discussed the resulting labeling. In all the questionable cases, the authors had a discussion, until a perfect consensus was reached. This was done for all open-ended questions.

4.1 Familiarity with Refactorings

We started the main body of the survey with the basic **Question 3** that asked how often the respondents performed any code refactoring recently. Figure 1 shows the breakdown of the answers. It can be seen that refactoring is an omnipresent practice in software development: 40.6% of developers indicated that they refactor code *Almost every day* and 36.9% more said that they refactor code *Every week*. Only about 20% of users said that they refactor code *Once or twice a month*, and just 2.5% *Never* refactored code.
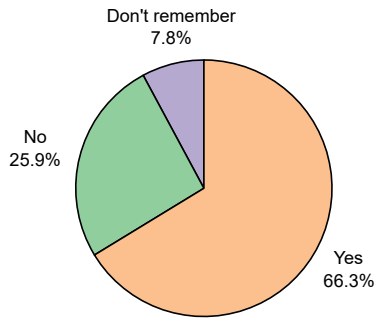

Figure 1: The answers to Question 3: *In the past month, how often have you performed any code refactoring?* (Out of 1,181 respondents)

However, it might be the case that developers may have a different perception of the word *refactor*. To account for this, if the participant selected *Once or twice a month* or *Never* in the previous question (**266** respondents), we additionally showed them **Questions 4** and **5**, asking whether they renamed or moved anything in the code recently. We found out that 84.2% of these 266 respondents renamed entities in the code and 78.2% of them moved code elements. Overall, only 14 respondents did not answer positively to either of these questions, and therefore, we did not consider their answers further, because they are not a target audience for any questions about refactorings. The remaining survey considered the answers of all the remaining 1,167 participants.

Respondents who are familiar with refactoring (**1,145** respondents) were exposed to the last question in this block, **Question 6**, asking whether they have been recently refactoring for an hour or more in a single session. The results are presented in Figure 2. Surprisingly, almost *two-thirds* of developers answered positively.

Figure 2: The answers to Question 6: *During this time, did you ever refactor code for an hour or more in a single session?* **(Out of 1,145 respondents)**

While it is widely accepted that refactoring helps to enforce better design practices or to cope with design defects [12], recent studies have shown that developers interleave refactoring activities with other maintenance and development-related tasks in practice, including feature updates, bug fixes, and API types migration [2, 4, 16, 23, 28, 38]. Yet, little is known about the overhead that refactorings exhibit, especially when they are executed in conjunction with these development tasks. According to the survey results, the majority of developers state that they spent over an hour while refactoring their code, which is interesting, considering that refactoring tools were designed to be executed independently, applying small edits. This finding was interesting to us, and we plan to monitor how refactoring features are being used for such time frame in the future. Researchers should also pay close attention to such refactoring sessions and study whether our current refactoring tools remain efficient during continuous refactoring.

Summary: *Intuitively, refactorings are a key element in the software development cycle, according to the participants of the survey. Nearly four out of five developers indicated that they refactored code every week or even almost every day recently. Interestingly, two-thirds of respondents said that they had refactoring sessions of an hour or longer during this time.*

4.2 Refactoring Approaches

The next section in the survey aimed to analyze how developers execute their intended refactoring, and the degree of their reliance on the IDE tools to do that. The results of survey questions corresponding to this investigation are presented in Figure 3.

The first question (**Question 7**) aimed to verify whether developers refactor code using built-in automated tools of IDEs or they perform it manually. First, we clustered refactorings into 3 main categories, namely the *Rename*, the *Extract*, and the *Move* categories. Each category can target various code entities (*e.g.*, rename class, method, attribute, etc.). Then, we designed our question to let respondents choose for each refactoring category whether they refactor their code using the appropriate IDE feature, or they rely on intuitive programming practices, such as *Copy and Paste* or *Find and Replace*. Since developers may select more than one choice, we report the percentage of developers selecting each option for each refactoring category. All percentages are showcased in Figure 3a.

In general, it can be seen that a significant number of respondents use IDE features to perform refactorings. *Rename* is in the big lead, with as much as 85.8% of participants saying that they used the IDE feature for it. This is a strong indicator that *Rename* refactorings are implemented well as an IDE feature, their usage is intuitive, and they produce stable, reliable output. However, since renaming is a relatively simple change (at least compared to other refactorings), almost half the respondents also used the *Find and Replace* feature to locate all instances of a given code element and rename it. Also, *Rename* can be seen as the most universal of all the studied refactorings, because virtually no one said that they did not encounter a scenario for its use.

As for the *Extract* refactoring, significantly fewer people used the IDE feature, about 54.7%, especially when compared to the RENAME feature. However, the EXTRACT feature is still solicited by more than half of the respondents. Also, 10.7% of the respondents indicated that they did not have a scenario where they had to extract something. These findings are in line with the recent study of Alcocer et al. [1] where the authors reported the existence of certain usability issues of the *Extract Method* refactoring when using IntelliJ IDEA.
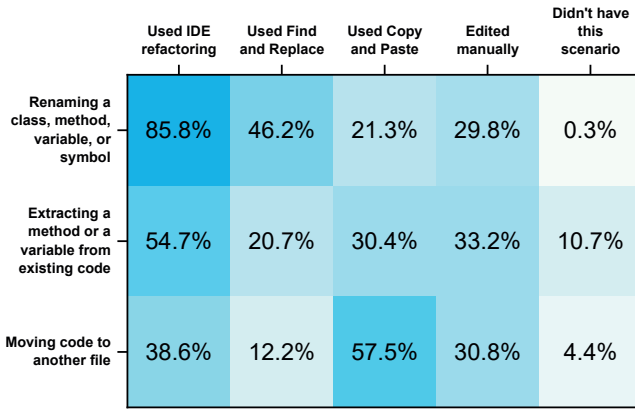
Finally, *Moving* code appears to be the first refactoring where the use of the IDE feature is not the most popular answer. 38.6% of developers answered that they use the IDE feature and more than half the respondents (57.5%) answered that they simply *Copy and Paste*. Occasionally, moving elements can be basic, and in this case, simply copying and pasting suffices. However, when the move is more difficult, when it involves dependencies and complex relationships between objects, then the IDE feature might be simpler to use. Still, in general, *Move* refactorings are almost as popular as *Renames*, with only 4.4% of participants saying that they did not encounter this scenario.

It is also alarming to notice that for all three refactorings, about one-third of respondents performed the refactoring by manual editing (typing).
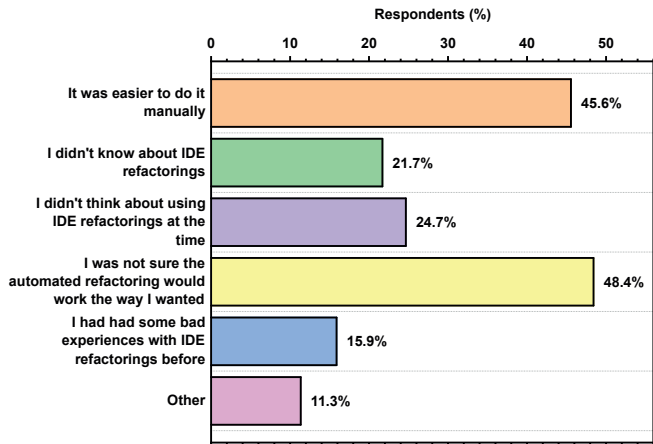
In the next **Question 8**, we selected all respondents who answered *Used Find and Replace*, *Used Copy and Paste*, or *Edited manually* for at least one refactoring (**1,014 respondents**), and asked them about the reasons for not using the IDE refactoring features. The results are presented in Figure 3b.

Two answers are the most popular. 48.4% of respondents said that they were not sure that the automated refactoring would work the way they wanted. This falls in line with the popular notion that developers often do not trust automated refactoring tools [6, 7, 23, 35, 39]. Also, 45.6% of respondents said that certain refactorings were easier to conduct manually. This puts the results of the previous question into perspective: a lot of developers use "manual" ways of conducting refactorings, because sometimes it is just more straightforward to do so. Therefore, one of the main takeaways for our IDE development teams is the importance of making the refactoring tools simpler and more intuitive. Since the RENAME feature is the most successful one in terms of adoption, analyzing how developers activate it would help in understanding its success, and potentially replicating it to other types of refactorings.
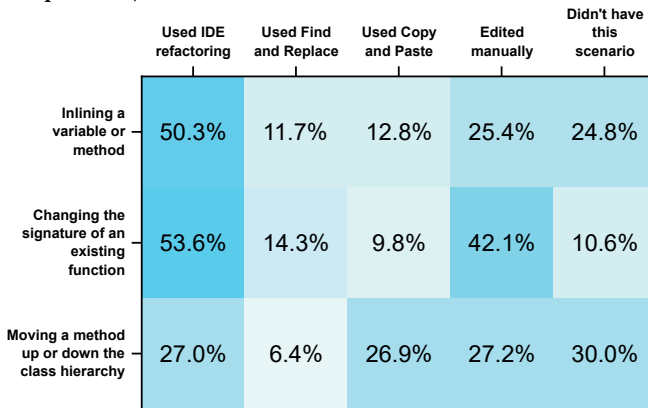
Fewer participants selected other options: 24.7% said that they did not think about using IDE refactoring tools at the time, 21.7% said that they did know about the IDE tools, and 15.9% said that
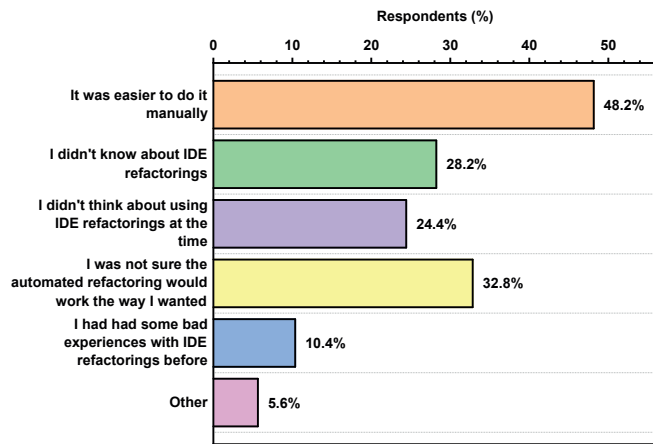
	Used IDE refactoring	Used Find and Replace	Used Copy and Paste	Edited manually	Didn't have this scenario
Renaming a class, method, variable, or symbol	85.8%	46.2%	21.3%	29.8%	0.3%
Extracting a method or a variable from existing code	54.7%	20.7%	30.4%	33.2%	10.7%
Moving code to another file	38.6%	12.2%	57.5%	30.8%	4.4%

(a) The answers to Question 7: *For the following scenarios, please select all the approaches you have used in the past month.* Several approaches can be selected for each refactoring, the percentages show the proportion of respondents who chose each option. (Out of 1,167 respondents)

(b) The answers to Question 8: *In these scenarios (Renaming, Extracting, Moving), what were your main reasons for not using the IDE refactoring feature?* (Out of 1,014 respondents)

	Used IDE refactoring	Used Find and Replace	Used Copy and Paste	Edited manually	Didn't have this scenario
Inlining a variable or method	50.3%	11.7%	12.8%	25.4%	24.8%
Changing the signature of an existing function	53.6%	14.3%	9.8%	42.1%	10.6%
Moving a method up or down the class hierarchy	27.0%	6.4%	26.9%	27.2%	30.0%

(c) The answers to Question 9: *For the following scenarios, please select the approaches you've used in the past month.* Several approaches can be selected for each refactoring, the percentages show the proportion of respondents who chose each option. (Out of 1,167 respondents)

(d) The answers to Question 10: *In these scenarios (Inlining, Changing the Signature, Pulling/Pushing), what were your main reasons for not using the IDE refactoring feature?* (Out of 868 respondents)

Figure 3: Answers to Questions 7–10 about different ways of using refactorings.

they had certain negative experiences with refactoring features before. When given a prompt to answer freely, some respondents mentioned several issues with refactoring features: performance issues, namespace confusion, difficulties with refactorings that involve significant changes to the logic of the program. This feedback is critical to our IDE development teams, and therefore, there will be a follow up with these respondents to seek more technical details with regard to their issues.

After this, we repeated the same line of questioning in regards to three other refactorings, namely, *Inlining* a variable or a method, *Changing the signature* of a function, and *Moving a method up and down the class hierarchy* (known also as *Pull up* and *Push down*). The answers to **Question 9** are demonstrated in Figure 3c.

It can be seen that *Changing the signature* is approximately as popular as *Extracting* in **Question 7**, with 10.6% of respondents saying that they did not recently have this scenario in their work and 53.6% saying that they used automated IDE tools. For this

refactoring, editing manually was also a very popular answer. As for the other two refactorings, they seem to be less popular than the previous ones. For *Inlining*, a quarter of the respondents said that they did not recently encounter this scenario in their work. Still, half the respondents said that they use IDE features for *Inlining*. *Pulling up / Pushing down* appears to be the least popular of the chosen refactorings. 30% of respondents did not recently encounter this refactoring in their work, and only 27% used IDE features for these refactorings.

In **Question 10**, we once again selected participants who answered *Used Find and Replace*, *Used Copy and Paste*, or *Edited manually* for at least one refactoring from **Question 9** (**868 respondents**) and asked them about their reasons for not using the IDE tools. The results are presented in Figure 3d. The general distribution of answers is similar to Figure 3b, but there are some differences between them. There are more answers *It was easier to do it manually* and *I did not know about IDE refactorings*. It is possible that

	Don't know about it	Know about it, but don't use it	Used it in the past, but will not use it again	Use it sometimes	Use it regularly
Rename file, class, method, symbol, etc.	5.1%	4.5%	2.1%	22.6%	65.7%
Extract method, variable, component, etc.	22.5%	18.8%	2.7%	25.7%	30.3%
Move	27.4%	15.7%	3.4%	27.5%	26.0%
Inline variable or method	30.3%	18.9%	2.1%	25.3%	23.4%
Change signature	26.8%	15.0%	2.7%	27.0%	28.5%
Pull Up / Push Down member	42.8%	20.7%	2.7%	19.3%	14.5%

Figure 4: The answers to Question 11: *How familiar are you with the following IDE refactoring features?* **(Out of 1,167 respondents)**

these refactorings are less universally known. Another takeaway for our IDE development teams is to attract developers' attention to the possibility of using these refactorings, perhaps by implementing special tooltips or IDE notifications.

> **Summary:** *Renames are the most universal refactoring, with virtually everyone using them. 85.8% of participants renamed elements using IDE tools. On the other side, 30% of respondents stated that they did not recently perform Pull Up and Push Down refactorings. Despite the existence of IDE refactoring features, developers still manually refactor their code, including Find and Replace for Rename, Copy and Paste for Move, Editing manually for Changing signature. Respondents justified their reluctance to use IDE refactoring features with not knowing the outcomes, along with manual refactoring being more intuitive.*

4.3 IDE Refactoring Features

In this section, we focused our questions specifically on IDE refactoring features. **Question 11** asked the developers about their familiarity with IDE refactoring features for the same six refactorings studied in Section 4.2. The heatmap with all the results is demonstrated in Figure 4.

Our first observation from the figure is the middle column. For all the refactorings, there were equally very few respondents who said that they refactored in the past, but will not do it again. This is a very positive insight meaning that even with all the concerns raised about automatic refactoring features, developers generally do not give up on them.

Coming to the other answers, they correlate fairly well with the previous questions. Once again, we can see the same two main outliers. RENAME is the most popular refactoring feature, with 65.7% of respondents saying that they use it regularly and 22.6% more saying that they use it sometimes. On the other hand, only 5.1% of participants said they are unaware of the RENAME IDE feature. Meanwhile, PULL UP / PUSH DOWN remains the least popular: up to 42.8% of developers did not know about the PULL UP / PUSH DOWN IDE feature, only 19.3% of respondents used it sometimes, and only 14.5% used it regularly. All the other automated refactoring features

(EXTRACT, MOVE, INLINE, and CHANGE SIGNATURE) are distributed more similarly between RENAME and PULL UP / PUSH DOWN.

1,078 respondents (92.4%) selected *Use it sometimes* or *Use it regularly* for at least one refactoring feature. For these participants, the next block of questions was unlocked. In **Question 12**, we asked the participants about their overall experience when using the last several IDE refactoring features. The results are presented in Table 2.

Table 2: The answers to Question 12: *Please think about the last several times you used IDE refactorings. How happy were you with the overall experience?* **(Out of 1,078 respondents)**

Answer	% of respondents
Not at all happy	0.3%
Not happy	1.6%
Neither happy nor unhappy	12.6%
Happy	56.9%
Very happy	28.6%

It can be seen that the overall experience is overwhelmingly positive, with 85.5% of developers giving positive responses, and as much as 28.6% saying that they were *Very happy* with the last several uses of IDE refactoring features.

Still, even though only 1.9% of developers answered either *Not happy* or *Not at all happy*, there were 12.6% respondents who answered *Neither happy nor unhappy*. It is very important to focus on developers who do not give positive answers to get a deeper understanding of underlying shortcomings in IDE refactoring features. To do this, in **Question 13**, we asked all the respondents who did not give a positive answer to share their experience in an open form. Several issues were brought up by several participants:

(1) 22 developers mentioned that their main problem was a certain negative experience when a refactoring was performed inaccurately. 6 developers specifically mentioned that a refactoring broke the code or introduced new errors.

(2) 7 developers mentioned that using an IDE refactoring tool might be slower than doing the refactoring manually. On the other hand, for very large projects, the problem of correctly taking care of all namespaces for various variables and functions becomes more difficult.

(3) 3 developers mentioned that IDE refactoring tools have a steep learning curve for them. As seen in previous questions, almost all developers understand what Rename is, but may never use some more specific features such as Pull up / Push down simply because they have no environment to understand what they do and what they are for. 5 more developers mentioned that the existing features that they know are confusing for them.

(4) Related to the last point, 5 developers mentioned that they would want a more direct and more visible representation of IDE refactoring tools: special tips, or maybe a notification when a refactoring is possible.

(5) 3 developers mentioned a general lack of trust in all automated tools.

(6) 2 developers brought up another interesting issue: applying an automated refactoring can violate a specific coding style guideline or formatting. Therefore, even if the refactoring is done correctly, it might still need some editing afterward.

This list is not comprehensive, but it can serve as a blueprint to get an idea of the current challenges that developers are facing with regard to the usage of automated IDE refactoring tools.
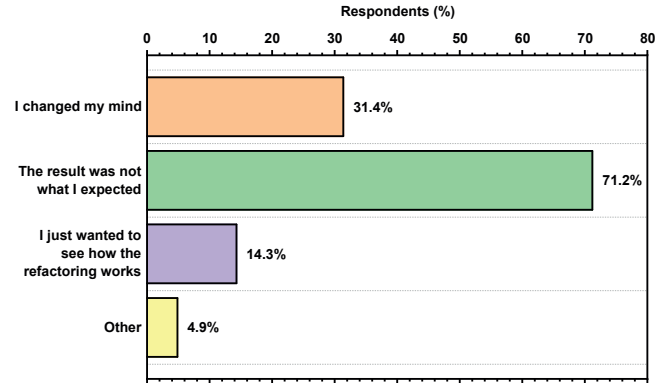
Similar to other IDE features, refactorings can be reverted in case their outcome seems unexpected to the developer. To target this event, we asked **Question 14** about how often developers had to undo a refactoring performed by the IDE. In order for the answers not to be vague, we specified the approximate percentage of time for each answer. The results are presented in Table 3.

Table 3: The answers to Question 14: *How often do you undo or revert an IDE refactoring action because you're unhappy with the result?* **(Out of 1,091 respondents)**

Answer	% of respondents
Often (>75% of the time)	3.0%
Every other time (≈50% of the time)	6.2%
Sometimes (≈25% of the time)	32.3%
Rarely (≈5% of the time)	50.8%
Never	7.7%

Naturally, it can be seen that in this question, extreme answers were rare. Indeed, only 7.7% of respondents said that they *Never* revert, but at the same time, only 6.2% said that they do it *Every other time (≈50% of the time)* and only 3% said that they do it *Often (>75% of the time)*. The vast majority of our respondents lied in between these poles, with 32.3% saying they undo refactoring actions *Sometimes (≈25% of the time)* and half of all participants (50.8%) saying they do it *Rarely (≈5% of the time)*.

In general, Undo is one of the most fundamental actions in programming, it happens all the time for various reasons. In a way, it is an inherent part of the creative process. Nevertheless, it is captivating for us to discover the reasons behind developers

Figure 5: The answers to Question 15: *The last few times you undid or reverted an IDE refactoring feature, what were the reasons?* **(Out of 1,007 respondents)**

undoing IDE refactoring actions. We asked this in **Question 15** of all the respondents who did not choose *Never* in **Question 14** (**1007** respondents). The answers are presented in Figure 5.

The most popular answer by far is that the produced result of the IDE feature was not what the developer expected, with 71.2% of participants selecting this option. 31.4% of respondents said that they changed their mind, and also 14.3% said that they just wanted to see how the refactoring would work. Among other reasons that the developers shared were their own mistakes, mishaps, and wrong configuration of refactoring (like namespaces of formats).

Summary: *85.5% of the respondents said that their experience with the IDE refactoring features is positive. A deeper inquiry about the existing challenges resulted in a list of possible issues to consider, including the difficulty of refactoring large projects, and the learning curve of some refactoring types. The vast majority of developers occasionally undo or revert refactorings, with the main reason being that the refactoring produced the result that was unexpected.*
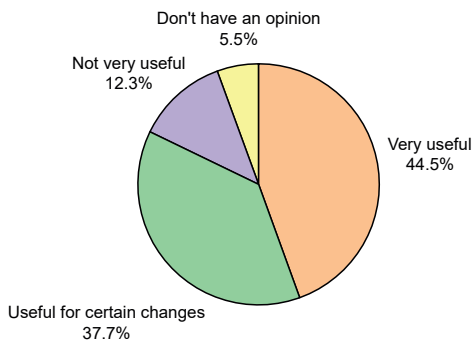
4.4 Previewing Refactorings

Previewing the outcome of refactoring before its execution is one of the main features of IntelliJ IDEs. Previously, in **Question 13**, some developers mentioned that their reliance on the refactoring feature highly depends on the preview, and that for some of them, the preview makes the biggest difference. We decided to investigate this further, so our next block of questions is aimed specifically at the process of previewing the refactoring.

In **Question 16**, we asked all the respondents whether they find the previewing useful. The distribution of the results is presented in Figure 6. The opinion about previewing refactorings is also largely positive. 44.5% of the respondents said that the previews are *Very useful*, and 37.7% said that they are *Useful for certain changes*. Overall, this constitutes more than 80% of participants. On the other hand, 12.3% said that the previews are *Not very useful* and 5.5% more did not have an opinion on the subject.

While this in itself proves the importance of the previewing function, it is of interest to dive deeper into both sides of this question. For this reason, we divided the respondents into those

Figure 6: The answers to Question 16: *When refactoring the code, do you find it useful to preview all of your changes before applying them?* **(Out of 1,167 respondents)**

who gave positive answers and those who did not, and asked them separate questions.

For participants who answered positively, **Question 17** asked for specific types of refactorings, for which they find the preview to be especially useful. The distribution of answers correlated well with the overall distribution of the popularity of the refactoring feature usage (as indicated by the last two columns in Figure 4), so it might be the case that the preview is used more or less similarly for different refactorings, at least no obvious anomalies were present.
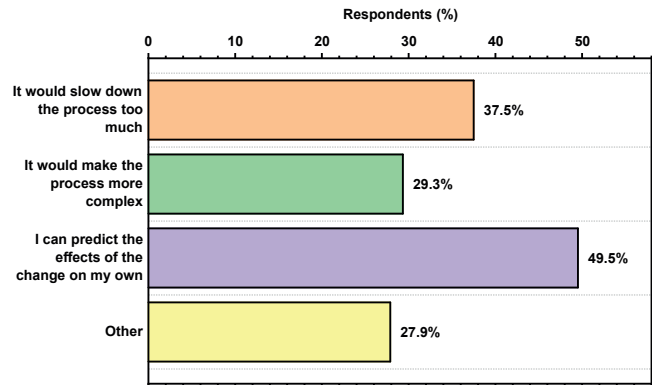
Then, in **Question 18**, we asked the respondents about their reasons for wanting to see the preview. The most popular answer, indicated by as many as 227 developers, is making sure that everything is done as they want it to be, with 22 developers specifically saying they are making sure that the code does not break, and 13 developers mentioning using this to combat a distrust in automatic features. However, other specific aspects were brought up.

(1) 78 developers specifically mentioned making sure that the IDE will not do anything extra and checking the impact of the changes on the whole codebase. Often, this has to do with renaming, and developers check namespaces, imports, and occurrences in comments.

(2) 8 developers check the preview carefully because they say it prevents them from wasting time later to check the Version Control System (VCS) diff or undo the unwanted changes.

(3) 5 developers said that they check the readability of the code during the change, making sure that applying refactoring will not worsen it, as well as check code style. They mentioned that in this regard, the visual aspect of the preview is very important.

(4) One developer mentioned that they are interested in the preview as a means to comprehend their code. Basically, they look at the preview not to check if the IDE missed something, but rather to check whether *they* forgot something. Several respondents mentioned that the preview allows them to think about the necessity of the change one more time.

It can be seen that apart from its main purpose, the preview sometimes serves different other goals. We also observed that the reasons for seeing the preview correlate well with the general shortcomings of refactoring features that developers mentioned in Section 4.3. The preview can elevate many of the concerns that have

to do with refactorings, even such specific ones as, for example, the uniformity of code style.

It is also important to understand the reasoning of participants who did not answer positively to the question about the usefulness of the preview. To such respondents, we showed **Question 19** asking them for their reasons. The distribution of the results is shown in Figure 7.

Figure 7: The answers to Question 19: *What are your main reasons for not finding a preview useful?* **(Out of 208 respondents)**

It can be seen that almost half of the respondents who were asked this question selected the answer *I can predict the effects of the change on my own*. The other responses are a little less popular: the previewing slowing down the IDE too much or adding too much to the complexity of the process.

When given a prompt to write the *Other* answer, respondents mentioned several reasons.

(1) 13 developers said that they prefer to use the VCS diff post-factum, motivating it with the fact that it produces the same functionality but also merges it with the possibility to verify a wider set of changes.

(2) 10 developers were simply very different from the ones in the previous questions, saying that they would rather simply run the code, and if it fails, they can always undo the change.

(3) 6 developers mentioned that the preview function is too complex for them or confuses them.

(4) 5 developers did not know about or did not encounter the preview feature.

In general, it seems that the opinion strongly depends on the developers, and in our survey, more developers demonstrated a cautious approach.

Summary: *Previewing is a critical feature for the IDE refactoring process. 82.2% of our respondents found it useful at least for specific changes. Apart from simply making sure that the refactoring is what is intended, it is useful in other situations, such as a fail-safe against a fast decision, a reason to review the change, as well as a verification for code readability and compliance with the code style of the project. A minority of our respondents were not positive about the preview, with half of them justifying this with being able to predict the refactoring outcome without the preview.*

4.5 Final Thoughts

The last question in our survey, **Question 20**, was an open-ended question for describing general thoughts about refactorings and IDE refactoring features. In general, the respondents' messages were positive. 177 developers expressed their compliments, respondents very often mentioned that refactorings are a cornerstone of the development process, and for some of them — the main reason to use an IDE in the first place. Let us enumerate the key findings in their feedback.

Refactoring knowledge. 52 developers reiterated a previous point about refactoring tools having a steep learning curve. They suggested that refactorings should be directly enabled in IDEs, through tips and popups. Some developers even suggested having interactive tutorials within the IDE that show up for first-time usage. Interestingly, several respondents mentioned that they are quite familiar with some refactoring types and hardly know anything about others.

Language difference. 9 developers commented on how refactorings behave differently in different languages. The biggest difference is noted between statically typed languages like Java and dynamically typed languages like Python, with refactorings in the latter ones being much less predictable. This is understandable from the nature of the language itself, however, tool developers should pay closer attention to refactorings in dynamically typed languages. This supports the findings in other works [24, 30] observing that developers are interested in refactorings for more languages, including the dynamically typed ones.

Scope. As mentioned before, one common problem the respondents have with refactorings is how they treat complex namespaces. Developers also mentioned the ability for any refactoring to be easily undone as a useful feature. Furthermore, developers suggested a more explicit and clear way to mark certain source files or code elements as a no-go zone for refactorings.

Complex features. Finally, several developers mentioned their thoughts on the complexity of tasks that refactorings solve. Some developers expressed the idea that most of the automatic refactorings, like the ones we study in this survey, are useful and time saviors, but they represent basic atomic actions. They suggested supporting more complex features like splitting a large complex method or class into several smaller ones. On the other hand, several developers mentioned that for simpler changes, it is often much faster to apply the refactorings manually, and that automated refactorings do not need to strive to replace the simpler changes completely.

5 THREATS TO VALIDITY

A large percentage of our participants (48.1%) have more than 10 years of professional experience. Moreover, the respondents might be more experienced with refactorings because IntelliJ-based IDEs are known for their rich support of refactorings. In our questionnaire, we focused only on a part of refactorings the IntelliJ-based IDEs support, which are the most well-known and studied [23]. However, in the *Final Thoughts* section of the questionnaire, the respondents were free to provide feedback about their experience with any refactoring.

Our survey involved participants who use a large variety of languages and IDEs, which could influence the way they plan their refactorings. We did not classify our findings by a specific language or an IDE. Further research aimed at investigating the possible differences is necessary.

Since all participants in our survey are paid subscribers of Jet-Brains IDEs, our results may not generalize to other contexts and other companies. Also, concerning the correctness of our interpretation of open responses, we did not discuss all responses because some of them are open to various interpretations, and follow-up surveys or interviews are needed to clarify them.

6 IMPLICATIONS

Educating about refactorings. The results show that developers are not familiar with some of the refactoring types that IDEs support (see Figure 4). Also, some developers might be cautious about the side-affects of refactoring, so they are not likely ready to perform any refactoring unless they get to know how it works (see Figure 3b). To encourage users to use automatic refactorings tools in IDEs, it might be useful to start with educating users about them. For example, along with showing a possibility to perform a refactoring, IDEs could provide some information about its purpose as well. The knowledge about how refactoring works will help IDEs to respond to user expectations (see Table 3 and Figure 5).

Increasing awareness about refactoring possibilities. One of the reasons the developers do refactorings manually is that they do not know about the possibility to perform them automatically [35]. It could be helpful if IDEs suggested possible refactoring opportunities for the user. The results show that users often refactor their code, sometimes for more than an hour, so automatic refactoring recommendations could save them time.

Supporting complex refactorings. Since some developers expressed their need in support of complex refactorings in IDEs, it would be exciting to support batch refactorings [5, 37], even if they are less intuitive. Such series of transformations would result in more atomic methods that better optimize structural metrics, such as complexity and lines of code. A lot of research efforts have focused on exploring the impact of refactoring on software quality using metrics [3, 8, 21, 27], and it would be interesting to turn this into actual IDE features.

7 CONCLUSION

In this paper, we presented the results of a large-scale survey about refactoring usage, conducted by JetBrains Research among the users of IntelliJ-based IDEs. The survey consists of several blocks of questions: familiarity with refactorings, using different approaches to conduct refactorings, making use of specific IDE refactoring features, and using previews when employing refactorings. Together with the background information at the start and some final thoughts at the end, the results of this survey represent a unique perspective of 1,183 experienced developers.

We hope that our results can be useful for both researchers and practitioners, and the presented opinions of more than a thousand developers can be used to perfect our methods and our tools.

REFERENCES

[1] Juan Pablo Sandoval Alcocer, Alejandra Siles Antezana, Gustavo Santos, and Alexandre Bergel. 2020. Improving the Success Rate of Applying the Extract Method Refactoring. *Science of Computer Programming* 195 (2020), 102475. https://doi.org/10.1016/j.scico.2020.102475

[2] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. (2021). https://doi.org/10.1109/ICSE-SEIP52600.2021.00044

[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the Impact of Refactoring on the Relationship Between Quality Attributes and Design Metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870177

[4] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. 2021. How We Refactor and How We Document It? On the Use of Supervised Machine Learning Algorithms to Classify Refactoring Documentation. *Expert Systems with Applications* 167 (2021), 114176. https://doi.org/10.1016/j.eswa.2020.114176

[5] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A Quantitative Study On Characteristics and Effect of Batch Refactoring on Code Smells. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870183

[6] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. Increasing the Trust In Refactoring Through Visualization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 334–341. https://doi.org/10.1145/3387940.3392190

[7] John Brant and Friedrich Steimann. 2015. Refactoring Tools Are Trustworthy Enough and Trust Must Be Earned. *IEEE Software* 32, 6 (2015), 80–83. https://doi.org/10.1109/MS.2015.145

[8] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 465–475. https://doi.org/10.1145/3106237.3106259

[9] Daniela S Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 international symposium on empirical software engineering and measurement*. 275–284. https://doi.org/10.1109/ESEM.2011.36

[10] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to advanced empirical software engineering*. 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[11] Developer Economics. accessed: 05.07.2021. State of the Developer Nation — 2021 Q1, https://www.developereconomics.com/resources/reports/. https://www.developereconomics.com/resources/reports/

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

[13] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. accessed: 05.07.2021. Supplementary Data Package, https://zenodo.org/record/4923175. https://zenodo.org/record/4923175

[14] Robert M Groves, Floyd J Fowler Jr, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. 2011. *Survey Methodology*. Vol. 561. John Wiley & Sons.

[15] Benedikt Hauptmann, Sebastian Eder, Maximilian Junker, Elmar Juergens, and Volkmar Woinke. 2015. Generating Refactoring Proposals to Remove Clones from Automated System Tests. In *2015 IEEE 23rd International Conference on Program Comprehension*. 115–124. https://doi.org/10.1109/ICPC.2015.20

[16] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding Type Changes in Java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 629–641. https://doi.org/10.1145/3368089.3409725

[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

[18] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal Opinion Surveys. In *Guide to advanced empirical software engineering*. 63–92. https://doi.org/10.1007/978-1-84800-044-5_3

[19] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[20] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[21] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 17. https://doi.org/10.1145/2729974

[22] Emerson Murphy-Hill and Andrew P Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE software* 25, 5 (2008), 38–44. https://doi.org/10.1109/MS.2008.123

[23] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18. https://doi.org/10.1109/TSE.2011.41

[24] Christian D. Newman, Mohamed Wiem Mkaouer, Michael L. Collard, and Jonathan I. Maletic. 2018. A Study on Developer Perception of Transformation Languages for Refactoring. In *Proceedings of the 2Nd International Workshop on Refactoring (IWoR 2018)*. 34–41. https://doi.org/10.1145/3242163.3242170

[25] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the Refactoring Mechanics. *Information and Software Technology* 110 (2019), 136–138. https://doi.org/10.1016/j.infsof.2019.03.002

[26] Jonhnanthan Oliveira, Rohit Gheyi, Felipe Pontes, Melina Mongiovi, Márcio Ribeiro, and Alessandro Garcia. 2020. Revisiting Refactoring Mechanics from Tool Developers' Perspective. In *Brazilian Symposium on Formal Methods*. 25–42. https://doi.org/10.1007/978-3-030-63882-5_3

[27] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving Code: The (Mis) Perception of Quality Metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 80–91. https://doi.org/10.1109/ICSME.2018.00017

[28] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-Based Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–30. https://doi.org/10.1145/3408302

[29] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering, Early Access* (2018), 146–164. https://doi.org/10.1109/TSE.2018.2884911

[30] Gustavo H Pinto and Fernando Kamei. 2013. What Programmers Say About Refactoring Tools? An Empirical Investigation of StackOverflow. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*. 33–36. https://doi.org/10.1145/2541348.2541357

[31] IntelliJ Platform. accessed: 05.07.2021. Open Source Platform for Building IDEs and Developer Tools, https://www.jetbrains.com/opensource/idea/. https://www.jetbrains.com/opensource/idea/

[32] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with Synthesis. *ACM SIGPLAN Notices* 48, 10 (2013), 339–354. https://doi.org/10.1145/2544173.2509544

[33] Max Schäfer and Oege De Moor. 2010. Specifying and Implementing Refactorings. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 286–301. https://doi.org/10.1145/1869459.1869485

[34] Tushar Sharma, Girish Suryanarayana, and Ganesh Samarthyam. 2015. Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. *IEEE Software* 32, 6 (2015), 44–51. https://doi.org/10.1109/MS.2015.105

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

[36] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2012. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2012), 147–162. https://doi.org/10.1109/TSE.2012.19

[37] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana C Bibiano, Daniel Oliveira, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 186–197. https://doi.org/10.1145/3379597.3387477

[38] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*. 483–494. https://doi.org/10.1145/3180155.3180206

[39] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson. 2012. Use, Disuse, and Misuse of Automated Refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 233–243. https://doi.org/10.1109/ICSE.2012.6227190

[40] Mohsen Vakilian and Ralph E Johnson. 2014. Alternate Refactoring Paths Reveal Usability Problems. In *Proceedings of the 36th international conference on software engineering*. 1106–1116. https://doi.org/10.1145/2568225.2568282