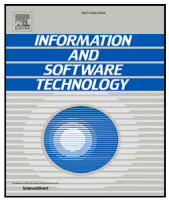
Information and Software Technology 162 (2023) 107273

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Towards a better understanding of the mechanics of refactoring detection tools

Jonhnanthan Oliveira [a,*], Rohit Gheyi [a], Leopoldo Teixeira [b], Márcio Ribeiro [c], Osmar Leandro [a], Baldoino Fonseca [c]

[a] Federal University of Campina Grande, Brazil
[b] Federal University of Pernambuco, Brazil
[c] Federal University of Alagoas, Brazil

ARTICLE INFO

Keywords:
Refactoring
Mechanics
Program comprehension

ABSTRACT

Context: Refactoring is a crucial practice used by many developers, available in popular IDEs, like Eclipse. Moreover, refactoring detection tools, such as RefDiff and RefactoringMiner, help improve the comprehension of refactoring application changes.
Objective: In this article, we better understand to what extent refactoring detection tools (RefDiff and RefactoringMiner) identify refactoring operations that developers apply in practice.
Methods: We survey with 53 developers of popular Java projects on GitHub. We asked them to identify six refactoring transformations applied to small programs.
Results: There is no unanimity in all questions of our survey. Refactoring detection tools do not detect many refactoring operations expected by developers. In 4 out of 6 questions, most developers prefer the Eclipse refactoring mechanics.
Conclusion: The results highlight the importance of diving deep into the refactoring mechanics and defining a baseline. Empirical studies focused on mining refactoring operations may be limited by an incomplete or unrepresentative sample of such operations, thus posing a challenge for researchers in this field.

1. Introduction

Refactoring [1] is the process of changing a program to improve its internal structure while preserving its observable behavior. IDEs, such as Eclipse and NetBeans, have automated many refactorings. Besides that, researchers have implemented tools [2,3] to identify refactoring applications considering step-by-step descriptions [1] of how to carry out the refactorings.

Nowadays, RefactoringMiner [2] and RefDiff [3] are the best refactoring detection tools available in the state-of-the-art. RefDiff can abstract programming language particularities, allowing it to support different languages, such as JavaScript, C, and Java. RefactoringMiner detects more than 40 refactoring types using an AST-based algorithm in the Java programming language. Identifying some refactoring operations in source code changes is valuable to understand software evolution, such as using refactoring tools, the motivations driving refactoring, the risks of refactoring, and the impact of refactoring on code quality metrics [3]. Refactoring detection tools help developers understand and review their code after a change scenario. Developers report a lack of tool support for refactoring change integration, code review tools targeting refactoring edits, and sophisticated refactoring engines in which a user can easily define new refactoring types bringing difficulties to integrate code changes after refactoring operations [4]. Vakilian et al. [5] identified factors that affect the developers' use of automated refactorings, making their output unpredictable. Furthermore, Oliveira et al. [6] found that most developers expect the refactoring output based on their experience. However, we need to find out to what extent refactoring detection tools identify refactoring operations that developers apply in practice.

In this work, we conduct a survey (Section 2) with 53 developers of popular Java projects on GitHub to better understand whether the mechanics of refactoring detection tools, such as RefactoringMiner [2] and RefDiff [3], are accepted by developers in practice. We asked them about the output of six refactoring types that are available in popular IDEs (Eclipse). We asked developers to identify the application of the Inline Method, Extract Method, Move Method, Push Down Method, Extract Interface, and Rename Class refactorings on small programs.

* Corresponding author.
E-mail addresses: jonhnanthan@copin.ufcg.edu.br (J. Oliveira), rohit@dsc.ufcg.edu.br (R. Gheyi), lmt@cin.ufpe.br (L. Teixeira), marcio@ic.ufal.br (M. Ribeiro), osmar@copin.ufcg.edu.br (O. Leandro), baldoino@ic.ufal.br (B. Fonseca).

https://doi.org/10.1016/j.infsof.2023.107273
Received 12 September 2022; Received in revised form 30 May 2023; Accepted 5 June 2023
Available online 16 June 2023

There is no unanimity in all questions of our survey. Few developers are aligned with the refactoring mechanics used by refactoring detection tools. In 4 out of 6 questions, most developers prefer the Eclipse refactoring mechanics. Refactoring detection tools do not detect many refactoring operations expected by developers. Previous studies have different results [5–8] for how developers apply refactorings. Researchers interested in mining refactoring operations are looking at an incomplete or even unrepresentative set of refactoring operations while doing their empirical studies. Moreover, this may impact developers' communication and misuse of complex refactorings with various parameters to customize a refactoring application in IDEs. Consider a team of developers discussing the impact of applying refactoring X to program Y. Since the mechanics are not precisely defined, each developer may have different ideas about refactoring mechanics. Other developers applying a refactoring X to Y might result in various programs (Y' and Y''). For instance, this scenario is worse when considering composite refactorings to change architecture.

The problem is that there are no complete mechanics specifications for all refactoring types that developers widely accept. Since there is no baseline, we cannot say which tool is correct. We decided to survey some developers to understand better the refactoring mechanics they expect. This may be one step towards defining a baseline and helping the refactoring community to know about this problem and work on how to solve it better.

2. Survey

We survey developers of popular Java projects on GitHub to understand whether developers follow the refactoring mechanics of refactoring detection tools in practice. To recruit participants, we sent e-mails to 1,000 developers randomly selected from popular GitHub projects, including Google, Facebook, and the Apache Foundation projects.

2.1. Planning

We divided the survey into three main sections. The first section asks developers which refactoring types were applied using six examples of transformations. We present a pair of programs illustrating a transformation and ask if any refactorings were applied. The output program in all questions can be yielded by applying a single refactoring of Eclipse. Developers were not aware of this. Each question shows a transformation relating to two Java programs with at most 18 LOC. We present four to six options. We include options indicating the refactoring mechanics used by RefactoringMiner and RefDiff. To yield their refactoring mechanics, we run RefactoringMiner and RefDiff on each transformation presented in a question and include an answer option for each output. We also include one option representing the Eclipse refactoring applied. Therefore the answer options extent may vary. All respondents answered the same questions in the same order. The second section asks for background information. Finally, the third section asks for additional comments.

Fig. 1 shows the first question of our survey. It presents a single application of the Inline Method refactoring using Eclipse and shows six options. The first option is aligned with the Eclipse refactoring mechanics. The second option is aligned with the RefDiff and RefactoringMiner refactoring mechanics. Our goal is to analyze the refactoring mechanics of refactoring detection tools to understand preferences from the viewpoint of developers in the context of small transformations. We address the following research question:

RQ$_1$ To what extent do refactoring detection tools identify refactoring operations that developers apply in practice?

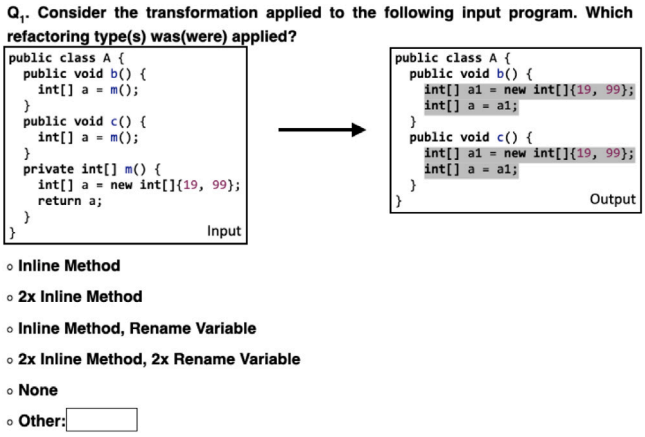


Fig. 1. First question of our survey.

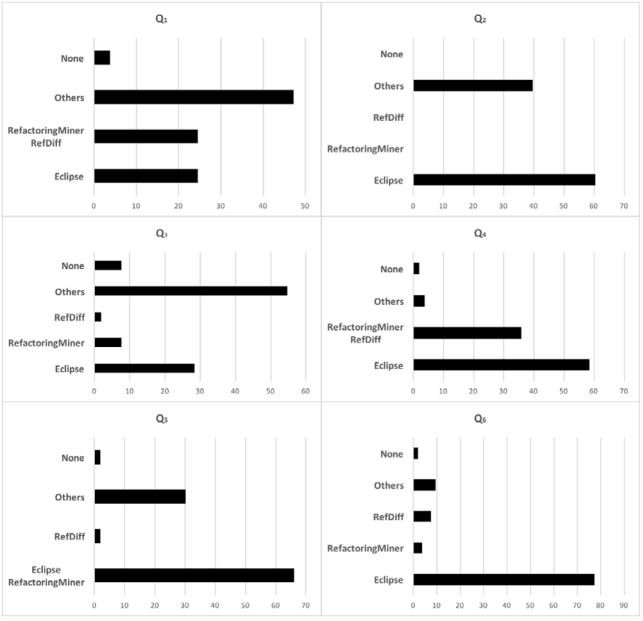


Fig. 2. Preferences of developers in each question.

2.2. Results and discussion

Fifty-three developers from different countries, e.g., Brazil, Canada, France, Germany, The Netherlands, Slovenia, Switzerland, Sweden, and the USA, completed the survey.[1] More than 30% of developers have more than 10 years of experience in refactoring programs, and more than 65% have 5 or more years of experience. In four out of six questions, more than 58% of developers are aligned with the Eclipse refactoring mechanics (Fig. 2).

[1] Survey: https://forms.gle/jGNTiCKtRyhceTYE6

Next, we explain the results of all questions in our survey. In Q_1 and Q_3, we show a single instance of the Inline Method and Move Method refactorings applied by Eclipse, respectively. Eclipse introduces a new variable in Q_1 and associates it with the original one. This causes RefactoringMiner to detect Rename Variable. Creating another variable in the same line as the older one causes misunderstanding when the developers see the differences in the code. Moreover, in Q_3, Eclipse changes the name and method parameter type. Because of this change, developers partially align with the detection performed by RefactoringMiner. Some developers are aligned with the Eclipse refactoring mechanics, while others are aligned with the RefactoringMiner or RefDiff refactoring mechanics. Less than 50% prefer other refactoring mechanics. Two other options of Q_1 (see Fig. 2) have almost the same preference. The community needs to discuss the refactoring mechanics. Q_2 presents a single instance of the Extract Method refactoring applied by Eclipse. RefactoringMiner and RefDiff are partially aligned with Eclipse. Refactoring detection tools detect the Extract Method refactoring and many other refactorings (e.g., Rename Variable). More than 58% of developers are aligned with the refactoring mechanics of Eclipse.

Listing 1: Multiple instances of the Push Down Method refactoring in RefactoringMiner and RefDiff.

```
@@ class Replacement
- public double normalizedEditDistance() { ... }
@@ class CompositeReplacement extends Replacement
+ public double normalizedEditDistance() { ... }
// 10 other subclasses changed
```

Q_4 presents a single instance of the Push Down Method refactoring applied by Eclipse. It is similar to the transformation shown in Listing 1. RefactoringMiner and RefDiff are partially aligned with Eclipse. If the superclass has N subclasses, the refactoring detection tools yield N instances of the Push Down Method refactoring. More than 55% of the developers agree with the refactoring mechanics of Eclipse, identifying a single application of the Push Down Method.

Q_5 presents a single instance of the Extract Interface refactoring applied by Eclipse. RefactoringMiner and RefDiff are partially aligned with Eclipse. Refactoring detection tools detect the Extract Interface refactoring and many other refactorings (Change Signature, Move Method, Pull Up Method, Change Parameter Type). More than 65% of developers agree with the refactoring mechanics of Eclipse.

Listing 2: Using the Eclipse Rename Class refactoring.

```
@@ class UMLJavadoc
- public class UMLJavadoc {
+ public class UMLDocJava {
@@ class UMLClass
- public void setJavadoc(UMLJavadoc javadoc) {
+ public void setJavadoc(UMLDocJava javadoc) {
```

Q_6 presents a single instance of the Rename Class refactoring applied by Eclipse. It is similar to the transformation presented in Listing 2. RefactoringMiner and RefDiff are partially aligned with Eclipse. Refactoring detection tools detect the Rename Class refactoring and many other refactorings (Change Parameter Type, Change Variable Type, Change Return Type, Change Attribute Type). More than 75% of developers agree with the refactoring mechanics of Eclipse. We submitted some of these transformations as an issue to RefDiff developers, but they did not answer yet.

RefactoringMiner developers may consider developers' opinions in previous questions to improve their tools by aligning with developers' preferences. For instance, based on the answers to Q_4, most developers expect one instance of the Push Down refactoring operation. Developers may consider improving the detection of Push Down refactoring to consider one instance instead of multiple instances.

The problem that our survey identifies is that each tool implements refactoring using its refactoring mechanics. Sometimes, this may not match developers' expectations, compromising the tool's usefulness. This may be why some developers still prefer to apply refactorings manually. Tempero et al. [9] speculate that an unstated barrier is difficult to translate a refactoring goal into refactoring operations. In this work, we further explore this barrier. One recommendation for those implementing refactoring operations is to show better each customization applied. This way, developers can better understand the refactoring mechanics before using the transformation.

Moreover, for those implementing refactoring detection tools, a recommendation is to explain the relationship of some refactorings better. For instance, a Rename Class refactoring may be associated with many other refactorings, such as Change Parameter Type (see Q_6). The refactoring mechanics used in refactoring detection tools may inflate the frequency of additional refactorings that are unexpected for developers who have only applied the primary refactoring in the IDE. Less than 10% of developers agree with RefactoringMiner or RefDiff. A better explanation of some refactorings' relationships may help interpret the results from such tools.

In summary, there is no consensus on all questions. Refactoring detection tools only detect some refactoring operations expected by developers. In 4 out of 6 questions, most developers follow Eclipse's refactoring mechanics. In Q_3, the majority of developers indicate a preference for refactoring mechanics different from the ones used by Eclipse, RefactoringMiner, and RefDiff. Researchers mining refactoring operations in software repositories should be aware of the limitation of refactoring detection tools, which only detect some or even identify an unrepresentative set of refactoring operations. Refactoring detection tools can improve their tools by considering developers' preferences. This may help researchers and developers better understand program evolution, increasing the popularity of these tools. Since we do not have a consensus on all questions, the results also highlight the importance of diving deep into the refactoring mechanics and defining a baseline.

3. Related work

Vakilian et al. [5] studied 26 developers working in their natural settings on their code for 1,268 programming hours over three months to understand how they interact with automated refactorings. They found that the interviewees did not know the mechanics of more than eight automated refactorings on average, and more than half could not describe the applied transformation. In our work, developers do not agree with some refactoring mechanics used by tools in practice.

Murphy-Hill et al. [10] find that some names of automated refactorings need to be clarified, and developers cannot predict the outcomes of complex tools. Even refactorings applied to small programs may lead to misunderstandings in identifying whether one or more refactorings were used.

Oliveira et al. [6] surveyed 107 developers of popular Java projects to understand their refactoring mechanics in practice better. They found that most developers expect the refactoring output based on their experience, and there is no unanimity in five out of seven questions in their survey. However, over 50% of the time, the IDEs used by developers yield an output that is different than if they manually apply the same refactoring. They found some differences. In our work, we survey with 53 developers and find more evidence of misunderstandings of the refactoring mechanics considering refactoring detection tools.

4. Conclusions

We surveyed 53 developers to better understand to what extent refactoring detection tools identify refactoring operations that developers apply in practice. Our survey has no unanimity, and most developers do not follow the refactoring mechanics used by refactoring

detection tools. This scenario may be even worse when considering coarse-grained refactorings applied to larger Java programs.

The misunderstandings explained in our work may be a starting point to improve refactoring detection tools. Improving refactoring implementations to detect a more extensive and representative set of refactoring operations that developers apply in practice may help increase their adoption. It may also help researchers mining software repositories to understand their results better about the motivations driving refactoring, the risks of refactoring, the impact of refactoring on code quality metrics, among other studies conducted using refactoring detection tools [2,3]. Since we do not have a consensus on all questions, our community needs to work on a refactoring mechanics specification widely used by developers. This may impact developers' communication and misuse of complex refactorings with various parameters to customize a refactoring application in IDEs [9].

In future work, we aim to include questions in the survey considering Java programs using more constructs, such as interfaces, abstract classes. We aim to dive deep into the step-by-step description of refactoring mechanics to identify improvement opportunities based on developers' preferences. This might make it easier for tool developers to follow them. We also intend to survey more developers and interview some to understand the problem better. For example, for a specific refactoring, considering changes in the accessibility constraints may be important for most developers. So, such constraints must be better detailed in the refactoring mechanics. We will select survey questions to which we do not have an alternative that the majority of developers chose to discuss during the interview, such as Q_1 and Q_3. During the interviews, we intend to dive deep into the refactoring mechanics and understand why they prefer one refactoring mechanics instead of others. We will show them real code examples and ask developers for real scenarios. Moreover, developers use other IDEs in practice, such as IntelliJ. In some cases, like in the transformation evaluated in Q_3, IntelliJ may have different refactoring mechanics than Eclipse. We intend to assess whether developers prefer IntelliJ's refactoring mechanics during the interviews. Based on their feedback, we aim to describe the refactoring mechanics' baseline in more detail. We also intend to analyze if the developers' preferences can be associated with specific factors, for instance, on coupling, cohesion, or other metrics. This way they might decide to either apply or avoid such changes. The baseline specification can be similar to those presented before [1] but must be complete. Considering all the programming language constructs, it will describe all changes that should and should not be performed for a given refactoring.

CRediT authorship contribution statement

Jonhnanthan Oliveira: Methodology, Investigation, Conceptualization, Data curation, Supervision, Writing – original draft. **Rohit Gheyi:** Methodology, Investigation, Conceptualization, Supervision, Writing – review & editing. **Leopoldo Teixeira:** Methodology, Investigation, Conceptualization, Supervision, Writing – review & editing. **Márcio Ribeiro:** Methodology, Investigation, Conceptualization. **Osmar Leandro:** Methodology, Data curation. **Baldoino Fonseca:** Methodology, Writing – review & editing.

Declaration of competing interest

There are no interests to declare.

Data availability

Data will be made available on request.

Acknowledgments

We want to thank the anonymous reviewers for their insightful suggestions. This work was partially supported by CNPq, Brazil, CAPES, Brazil, and FAPEAL, Brazil grants.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] N. Tsantalis, A. Ketkar, D. Dig, RefactoringMiner 2.0, TSE 48 (3) (2022) 930–950.
[3] D. Silva, J. Silva, G. Santos, R. Terra, M. Valente, RefDiff 2.0: A multi-language refactoring detection tool, TSE 47 (12) (2021) 2786–2802.
[4] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: FSE, 2012, pp. 50:1–50:11.
[5] M. Vakilian, N. Chen, S. Negara, B.A. Rajkumar, B. Bailey, R. Johnson, Use, disuse, and misuse of automated refactorings, in: ICSE, 2012, pp. 233–243.
[6] J. Oliveira, R. Gheyi, M. Mongiovi, G. Soares, M. Ribeiro, A. Garcia, Revisiting the refactoring mechanics, IST 110 (2019) 136–138.
[7] M. Kim, T. Zimmermann, N. Nagappan, An empirical study of refactoring challenges and benefits at Microsoft, IEEE TSE 40 (7) (2014) 633–649.
[8] E. Murphy-Hill, C. Parnin, A. Black, How we refactor, and how we know it, TSE 38 (1) (2012) 5–18.
[9] E. Tempero, T. Gorschek, L. Angelis, Barriers to refactoring, CACM 60 (10) (2017) 54–61.
[10] E. Murphy-Hill, M. Ayazifar, A.P. Black, Restructuring software with gestures, in: VL/HCC, 2011, pp. 165–172.