A Technique to Test Refactoring Detection Tools

Osmar Leandro
Federal University of Campina Grande
Brazil
osmarleandro@ufcg.edu.br

Rohit Gheyi
Federal University of Campina Grande
Brazil
rohit@dsc.ufcg.edu.br

Leopoldo Teixeira
Federal University of Pernambuco
Brazil
lmt@cin.ufpe.br

Márcio Ribeiro
Federal University of Alagoas
Brazil
marcio@ic.ufal.br

Alessandro Garcia
Pontifical Catholic Univ. of Rio de Janeiro
Brazil
afgarcia@inf.puc-rio.br

ABSTRACT

Refactoring detection tools, such as RefactoringMiner and RefDiff, are helpful to study refactorings applied to software repositories. To evaluate them, the tools' authors study software repositories and manually classify transformations as refactorings. However, this is a time-consuming and an error-prone activity. It is unclear to what extent the refactoring mechanics is consistent with refactoring implementations available in IDEs. In this paper, we propose a technique to test refactoring detection tools. In our technique, we apply a single refactoring using a popular IDE, and then we run the refactoring detection tool to check whether it detects the transformation applied by the IDE. We evaluate our technique by automatically performing 9,885 transformations on four real open-source projects using eight Eclipse IDE refactorings. RefactoringMiner and RefDiff detect more refactorings in 20.41% and 14.11% of the analyzed transformations, respectively. In the remaining cases, RefactoringMiner and RefDiff either do not detect the refactoring or classify it as other types of refactorings. We report 34 issues to refactoring detection tools, and developers fixed 16 bugs, and 3 bugs are duplicated. In other cases, 3 issues are not accepted. This study brings evidence for the need of a shared understanding of refactoring mechanics.

1 INTRODUCTION

Refactoring [9, 15, 24] is the process of changing a program to improve its internal structure while preserving its observable behavior. For a given refactoring, we use the term mechanics to denote the description of how to carry out such refactoring, as some works describe [9, 15, 24]. Popular IDEs, such as Eclipse, typically include a number of refactoring implementations. Moreover, tools that detect refactorings have been proposed in the literature. Currently, the best tools available in the state of the art are RefactoringMiner [37] and RefDiff [31]. Recovering refactoring information can provide useful insights to researchers focused on understanding software evolution. Some studies address important aspects related to refactorings, such as its motivations [19, 20, 39], improvements of detection algorithms [21, 26, 31, 37], understanding the perspective of developers [11, 19, 20, 22], and detecting behavioral changes introduced by refactorings [33, 34]. Knowing which refactoring operations were applied in the version history of a system may help in practical tasks, such as code reviews, proposing better diff visualization tools, and facilitating API library migration [31].

To evaluate the accuracy of refactoring detection tools, developers manually mine open-source projects to identify transformations based on their experience. Then, they create a dataset of manually classified transformations to evaluate their tools. For example, Tsantalis et al. [37] manually identified 7,226 refactorings in open-source projects for 40 different refactoring types. However, this process is time-consuming and error-prone. Since there is no refactoring mechanics specification widely accepted, refactoring detection tool developers may have different refactoring mechanics in mind [22]. It is unclear to what extent the refactoring mechanics is consistent with refactoring implementations available in popular IDEs.

In this paper, we propose a technique to test refactoring detection tools. In short, the technique consists of applying a refactoring using a popular IDE, and then running the refactoring detection tool to check whether it detects the transformation performed by the IDE. For instance, we apply the Move Method refactoring to a program using Eclipse. Then, we run a detection tool, such as RefDiff, to see whether it correctly detects the refactoring.

To evaluate our technique, we use the Eclipse IDE to apply 9,885 transformations of the following refactoring types: Rename Method, Rename Class, Move Method, Pull Up Method, Push Down Method, Extract Interface, Inline Method, and Extract Method. All of these refactorings are supported by *both* RefactoringMiner and RefDiff. This allows comparing the mechanics of these tools with the Eclipse IDE's. RefactoringMiner and RefDiff are aligned with the refactoring mechanics of Eclipse in 74.28% and 78.45% of the transformations, respectively. They detect more refactorings in 20.41% and 14.11% of the analyzed transformations, respectively. RefactoringMiner and RefDiff do not detect refactorings or detect other types of refactorings in other cases. We reported 34 issues to the developers of both tools, out of which 16 were fixed. At the moment, 12 bug reports are still open, 3 bugs are duplicated, and 3 issues are not accepted, which might indicate possible problems in the mechanics of refactoring implementations. Developers fixed bugs related to the Move Method (4), Inline Method (3), Rename Method (3), Extract Method (2), Extract Interface (2), Pull Up

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
@@ class RefactoringSet
- public getRevision() {
+ public getRevisionID() {
    return revision;
  } ...
```

Listing 1: Using the Eclipse Rename Method refactoring.

Method (1), and Push Down Method (1) refactorings. In summary, our main contributions are the following:

- A technique to test refactoring detection tools (Section 3);
- An evaluation comprising 9,885, which found 34 issues, out of which 16 are fixed (Section 4).

All experimental data are available online [12].

2 MOTIVATING EXAMPLE

Fowler [9] describes the mechanics of some refactorings. For instance, the Rename Method refactoring changes a method name to a new one that better reveals its purpose. Popular IDEs, such as Eclipse, implement this refactoring. Consider the `RefactoringSet` class declared in the RefactoringMiner source code. We can rename the `getRevision` method, changing its name to `getRevisionID`. This process can be automated using Eclipse. Listing 1 presents part of the actual transformation applied by Eclipse. Eclipse also performs similar transformations in all places where `getRevision` is used. The lines with `-` and `+` indicate lines of code that are removed and added, respectively.

We can use refactoring detection tools, such as RefactoringMiner and RefDiff, to detect refactorings applied by developers. If we use them to evaluate the transformation in Listing 1, RefactoringMiner 2.0.3 indicates that the Rename Parameter refactoring was applied, while RefDiff yields the Rename Method refactoring.

This example shows that the refactoring mechanics implemented by Eclipse is not consistent with the refactoring mechanics used by RefactoringMiner 2.0.3. In this particular example, we reported issue #141, and developers fixed it in RefactoringMiner 2.1.0.

The developers of refactoring detection tools create datasets to evaluate their tools. They manually mine open-source projects to identify refactorings based on their experience. For instance, Silva [30] uses a manual-validated dataset proposed by Tsantalis et al. [38], which contains non-refactoring and refactoring changes. However, this process is time-consuming and can be error-prone. It is unclear to what extent the refactoring mechanics of detection tools is consistent with refactoring implementations available in IDEs. Knowing the differences may help researchers using refactoring detection tools to better understand their results.

3 TECHNIQUE

Next we present our technique to test refactoring detection tools.

3.1 Overview

Figure 1 shows the main steps of our technique. It receives as input a program and a particular refactoring type, hereafter called `X`, to be evaluated. First, it searches for all possible locations where `X` can be applied in the input program. Then, for each location, using an implementation of `X`, it applies a *single* refactoring to the input program, producing a new version of the input program as output. Notice that this step yields a set of output programs containing the application of a single refactoring for each possible location identified in Step 1. Finally, we run a refactoring detection tool (`Y`) to check whether `Y` detects the application of refactoring `X` in each input and output program pair. The technique then produces a report indicating whether the refactoring detection tool was able to detect each refactoring application.

3.2 Steps

Step 1 consists of identifying all *locations* where we can apply a refactoring (`X`). In this step, the technique automatically searches for all possible refactoring targets in the input source code. We follow a similar approach as Gligoric et al. [10]. For example, suppose we would like to evaluate the Rename Class refactoring. Our technique finds all classes (locations) in the program received as input. As another example, consider we would like to evaluate the Rename Method refactoring. Our technique searches for all methods (locations) declared in the input program. The result of Step 1 is a set of locations (`L`). Locations are then dependent on the particular refactoring type. For instance, it might be a method call for the Inline Method refactoring, statements for the Extract Method refactoring, or a method declaration for the Move Method refactoring.

In Step 2, we apply the implementation of `X` to all possible locations `L`. Consider the Move Method refactoring as `X`. In Step 1, we find all method declarations in the input program. In Step 2, we apply the Move Method refactoring to each location, such as the `doHealthCheck` method from Listing 2. We move it from the `ElasticRestHealthInd` class to the `Health` class.

```
@@ class ElasticRestHealth
- doHealthCheck(builder, ...);
+ builder.doHealthCheck(this, ...);

- private void doHealthCheck(Health.Builder b, String json) {
- builder.withDetails(response); ...

@@ class Health
+ public void doHealthCheck(ElasticRestHealth e, String json) {
+ withDetails(response); ...
```

Listing 2: Using the Eclipse Move Method refactoring.

Listing 2 presents a pair of input and output programs. We repeat this process to all locations `L` identified in Step 1. The result of this step is a set of pairs (`P`) containing the input and output programs. We reinforce that each pair consists of an output program yielded by a single application of `X` to the input program. We do so to make it simpler to compare with the refactoring detection tool output. It also makes it easier to report issues to developers. If the refactoring implementation does not apply a transformation, or the resulting code does not compile, we ignore it.

In Step 3, we run the refactoring detection tool for each pair in `P`. For example, consider Listing 2 in which we apply the Eclipse implementation of Move Method to the `doHealthCheck` method. RefactoringMiner yields a single instance of the Move Method refactoring. For this pair, we then report that the refactoring mechanics of RefactoringMiner **aligns with** the refactoring implementation of Eclipse.

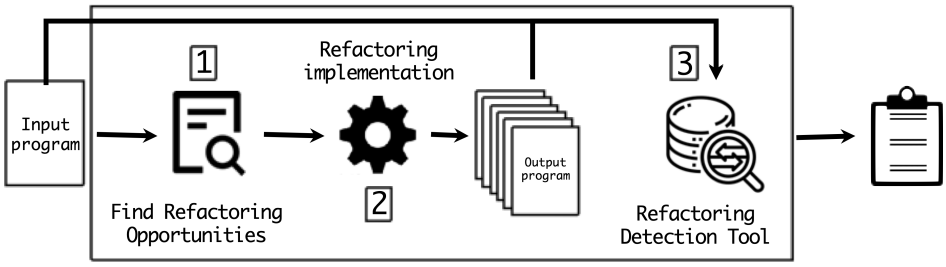

Figure 1: A technique to compare refactoring mechanics of refactoring detection tools and refactoring implementation.

Now, consider the example presented in Listing 3, in which we apply the Eclipse implementation of Rename Class to the `UMLJavadoc` class. For this pair, RefactoringMiner yields seven refactorings: Rename Class (1), Change Return Type (2), Change Attribute Type (2), and Change Parameter Type (2). For this pair, since the output of RefactoringMiner contains the refactoring applied by Eclipse amidst other refactorings, we report that the refactoring mechanics of RefactoringMiner **is partially aligned with** the refactoring implementation of Eclipse.

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

Listing 3: Using the Eclipse Rename Class refactoring.

As another example, suppose we now use Eclipse to apply the Rename Method refactoring to the `getRevision` method, changing its name to `getRevisionID`. We partially present this in Listing 1. Running RefactoringMiner on this pair yields the Rename Parameter refactoring. For this pair, since the output of RefactoringMiner is not the refactoring applied by Eclipse and the output is not empty, we report that the refactoring mechanics of RefactoringMiner **is different** from the refactoring applied by Eclipse.

In some cases, RefactoringMiner yields an empty output. Suppose we apply the Extract Method refactoring illustrated in Listing 4. RefactoringMiner 2.0.3 yields an **empty set** of refactorings, and does not detect the refactoring applied by Eclipse. We reported this problem to RefactoringMiner developers as issue #159, and RefactoringMiner 2.1.0 correctly detects the applied refactoring.

```
@@ class PrometheusPushGatewayManager
- String host = ex.getMessage();
- String message = ...
+ String message = extracted(ex);
@@ class PrometheusPushGatewayManager
+ private String extracted(UnknownHostException ex) {
+ String host = ex.getMessage(); ...
+ return message;
```

Listing 4: Using the Eclipse Extract Method refactoring.

Our technique classifies each pair into one of the four previously mentioned categories, and reports the results to the user. We summarize our categorization in Table 1. Our main goal is to detect differences between the refactoring mechanics implemented by IDEs and refactoring detection tools. When such a difference is found, it does not necessarily mean that the issue is on the refactoring detection tool side. Sometimes, the refactoring mechanics implemented by refactoring implementations may add some additional optional changes, such as introducing temporary variables when inlining a method. Some previous approaches found a number of bugs in refactoring implementations of popular IDEs [4, 10, 29].

Table 1: Classification of transformations. x = refactoring type applied by the refactoring implementation `A`; Y = the list of refactorings types detected by the refactoring detection tool `B`.

Category	Definition
`A` and `B` are different	$x \notin Y \wedge Y \neq \emptyset$
`A` is aligned with `B`	$x \in Y \wedge \#Y{=}1$
`A` is partially aligned with `B`	$x \in Y \wedge \#Y{>}1$
`B` yields an empty set	$Y = \emptyset$

3.3 Tool Support

For simplicity, we use a default parameter for other options available in the refactoring implementation. In Step 1, we use the Eclipse AST to perform code analysis and identify all possible locations where we might apply a refactoring, such as classes, methods, interfaces, fields, and so on. We use the refactoring implementations from Eclipse [6] in Step 2. So far, we have tool support for the following refactoring implementations: Rename Method, Rename Class, Move Method, Push Down Method, Pull Up Method, Extract Interface, Inline Method, and Extract Method, which are all of the refactorings that are present in Eclipse IDE and both of the refactoring detection tools assessed in this work. A similar approach can be used to include other refactoring implementations. Finally, in Step 3 we consider two refactoring detection tools currently, namely RefactoringMiner [37] and RefDiff [31].

4 EVALUATION

In this section, we use our technique to evaluate eight refactoring implementations of Eclipse using two refactoring detection tools in four open-source projects. We run two instances of the technique for each detection tool, first using Eclipse and RefactoringMiner, and next using Eclipse and RefDiff.

4.1 Study Definition

Our goal is to apply our technique to test refactoring detection tools, with the aim of finding mismatches between their refactoring mechanics and the ones implemented in Eclipse. We analyze RefactoringMiner, RefDiff, and the refactoring implementations of Eclipse. We address the following research questions:

- RQ_1 To what extent the refactorings applied by Eclipse are detected by RefactoringMiner or RefDiff?
  We count the number of refactorings detected by RefactoringMiner ($RQ_{1.1}$) or RefDiff ($RQ_{1.2}$) that are aligned with Eclipse, as well as the number of refactorings that are partially aligned and different. Finally, we also count the number of times that the detection tool yields an empty set.
- RQ_2 How many bugs can our technique detect in RefactoringMiner and RefDiff?
  We submit issues to the developers of refactoring detection tools and count the number of accepted and fixed bugs.

4.2 Experimental Setup

We ran the experiment on a laptop computer with Core i7 3.1 GHz and 8 GB RAM running Fedora 33 and Oracle JDK 1.8. Table 2 shows the four open-source projects used as inputs: Apache Gobblin, Google Maps Services Java, RefactoringMiner, and Spring Boot. All of them use Gradle[1] for build automation, and we used them in previous studies.

Table 2: Projects used in our evaluation.

Project	Domain	KLOC	Stars	Contributors
Apache Gobblin	A distributed data integration framework	454	2,100	90
Google Maps Services Java	A Java client for Google Maps Services	38	1,500	93
Spring Boot	A framework to create Spring-based applications	674	61,200	908
RefactoringMiner	A refactoring detection tool	127	237	13

We use eight refactoring implementations of Eclipse JDT 4.16 in Step 2. Eclipse is a widely used IDE and has a number of refactoring implementations. In this work, we consider eight refactoring implementations: Rename Method, Rename Class, Move Method, Push Down Method, Pull Up Method, Extract Interface, Inline Method, and Extract Method refactorings. In Step 3, we use two refactoring detection tools: RefactoringMiner 2.0.3[2] ($RQ_{1.1}$), and RefDiff 2.0[3] ($RQ_{1.2}$). To answer RQ_2, we analyze several transformations to see whether it is possible to arrive at the refactored version of the code by applying the detected refactorings. Then, we create an issue to discuss that behavior with the refactoring detection tool developers. All experimental data are available online [12].

For each refactoring implementation, we used the following parameters [12]. For the Extract Interface refactoring, we select all members of a class to declare in the interface. We use the class name with "I" prefix. For the Inline Method refactoring, we select a random method to inline. In the Pull Up Method refactoring, we select a random method to pull up in a subclass. In the Push Down Method refactoring, we select a random method to push down in a class that has a subclass. In the Rename Class and Method refactoring, we use the old class and method name with a suffix. For the Extract Method refactoring, we only apply it to methods that contain at least three statements. First, it tries to extract the second statement. If it cannot apply a refactoring, it tries to extract the second and third statements. We repeat this process by adding more statements until we successfully apply a refactoring using the IDE, or we reach the last statement. For the other parameters, we used the default value from Eclipse.

4.3 Results

Our technique analyzed a total of 9,885 transformations applied by Eclipse using eight refactoring types implemented by Eclipse, which were evaluated using RefactoringMiner and RefDiff. Table 3 summarizes our results.

We apply 2,740 transformations using the Rename Method refactoring implementation. RefactoringMiner and RefDiff are aligned in 2,696 (98.39%) and 2,380 (86.86%) transformations with Eclipse's refactoring mechanics, respectively. In five of the transformations, RefactoringMiner detects more refactorings. For instance, it detects the Change Variable Type and the Rename Parameter refactorings. In seven transformations, RefDiff detects other refactoring types. For example, RefDiff detects a combination of the Extract and Move Method refactorings instead of the Rename Method refactoring. Finally, RefactoringMiner and RefDiff do not detect any refactoring in 39 (1.42%) and 353 (12.88%) transformations, respectively.

We apply 643 transformations using the Rename Class refactoring implementation of Eclipse. RefactoringMiner and RefDiff are aligned in 399 (62.05%) and 523 (81.34%) transformations with Eclipse's refactoring mechanics, respectively. In 234 (36.39%) and 118 (18.35%) transformations, RefactoringMiner and RefDiff detect more refactorings. For instance, RefactoringMiner detects the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings. RefactoringMiner did not detect any refactoring in 7 transformations, while RefDiff failed to detect refactorings in 2 transformations. Finally, RefactoringMiner detects other refactorings in 3 transformations. For example, it reports the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings instead of the Rename Class refactoring.

We apply 1,558 transformations using the Move Method refactoring implementation. RefactoringMiner and RefDiff are aligned in 720 (46.21%) and 1,432 (91.91%) transformations with Eclipse's refactoring mechanics, respectively. In 669 (42.94%) transformations, RefactoringMiner yields more refactorings. For example, it yields up to 11 different refactoring types, such as the Add Parameter, the Change Parameter Type, the Inline Method and the Pull Up Method refactorings. RefactoringMiner and RefDiff detect other refactorings in 15 and 17 transformations, respectively. Finally, RefactoringMiner and RefDiff do not detect any refactoring in 154 (9.88%) and 109 (7%) transformations, respectively.

Using the Push Down Method refactoring implementation, we apply 145 transformations. Both tools are aligned with the Eclipse refactoring mechanics in 10 transformations. In 130 (89.66%) and

[1] https://gradle.org/
[2] https://github.com/tsantalis/RefactoringMiner/commit/fee2968
[3] https://github.com/aserg-ufmg/RefDiff/commit/2a06cfd

Table 3: Summary of the results. The second from the last column indicates the total number of refactorings applied by Eclipse JDT 4.16. The last column indicates the total number of reported issues. RM = RefactoringMiner; RD = RefDiff.

Refactoring	Tool	Aligned	Partially Aligned	Different	Empty	Total	Issues
Rename Method	RM	2,696	5	0	39	**2,740**	3
	RD	2,380	0	7	353		3
Rename Class	RM	399	234	7	3	**643**	1
	RD	523	118	2	0		2
Move Method	RM	720	669	15	154	**1,558**	4
	RD	1,432	0	17	109		4
Push Down Method	RM	10	130	0	5	**145**	2
	RD	10	134	1	0		2
Pull Up Method	RM	23	0	0	2	**25**	1
	RD	25	0	0	0		0
Extract Interface	RM	1,000	373	1	2	**1,376**	2
	RD	740	636	0	0		1
Inline Method	RM	263	396	13	229	**901**	4
	RD	503	346	0	52		1
Extract Method	RM	2,212	205	1	52	**2,470**	3
	RD	2,121	157	0	192		1

134 (92.41%) transformations, RefactoringMiner and RefDiff identify more refactorings. For example, they consider an instance of the Push Down Method refactoring for each subclass we push down instead of a single transformation. If we push down a method to five subclasses using Eclipse, the refactoring detection tools yield five instances of the Push Down Method refactoring. In 5 transformations, RefactoringMiner does not detect any refactoring. RefDiff detects the Move Method refactoring when we apply Push Down Method to a method that has a parameterized type replaced by a concrete type. We discuss this behavior in issue #16.

We use the Pull Up Method refactoring implementation to apply 25 transformations. RefactoringMiner and RefDiff are aligned in 23 (92%) and 25 (100%) transformations with Eclipse's refactoring mechanics, respectively. In 2 (8%) transformations, RefactoringMiner does not yield any transformation.

Using the Extract Interface refactoring implementation, we apply 1,376 transformations. RefactoringMiner and RefDiff are aligned with Eclipse's refactoring mechanics in 1,000 (72.67%) and 740 (53.78%) transformations, respectively. RefactoringMiner and RefDiff detect more refactorings in 373 (27.11%) and 636 (46.22%) transformations, respectively. For instance, RefactoringMiner detects the Change Variable Type and the Change Parameter Type refactorings. In addition, RefactoringMiner does not report any refactoring only in a single transformation. For example, it yields the Change Return Type refactoring, but it does not detect the Extract Interface refactoring. Finally, RefactoringMiner detects other refactorings in 2 transformations.

We apply 901 transformations using the Inline Method refactoring implementation. RefactoringMiner and RefDiff are aligned in 263 (29.19%) and 503 (55.83%) transformations with Eclipse's refactoring mechanics, respectively. RefactoringMiner and RefDiff detect more refactorings in 396 (43.95%) and 346 (38.40%) transformations, respectively. Besides, RefactoringMiner and RefDiff do not detect any refactoring in 229 (25.42%) and 52 (5.77%) transformations, respectively. Finally, RefactoringMiner detects other refactorings in 13 transformations. For example, it yields the Change Variable Type and Extract Variable refactorings instead of the Inline Method refactoring.

Finally, we apply 2,470 transformations using the Extract Method refactoring implementation. RefactoringMiner and RefDiff are aligned in 2,212 (89.55%) and 2,121 (85.87%) transformations with Eclipse's refactoring mechanics, respectively. RefactoringMiner and RefDiff detect more refactorings in 205 (8.30%) and 157 (6.36%), respectively. For instance, RefactoringMiner detects the Parameterize Variable, Rename Parameter and Rename Variable refactorings. Moreover, RefactoringMiner and RefDiff do not yield any transformation in 52 (2.11%) and 192 (7.77%) transformations, respectively. Finally, RefactoringMiner detects another refactoring in a single transformation only. For this case, it yields the Change Parameter Type instead of the Extract Method refactoring.

4.4 Discussion

In this section, we discuss the results of our technique.

4.4.1 Bug Reports. After the last step of our technique (Figure 1), we manually classify failures into distinct issues, which are then reported to developers. For each mismatch in a cluster, the first author took a few minutes to better understand it and check whether it was a real bug candidate to report to the refactoring detection tool. We analyze all transformations in the Partially Aligned, Different, and Empty categories (see Table 1).

We analyze the results of each refactoring detection tool separately and discuss the main issues in the following sections. For the Different and Empty categories, we randomly select one transformation to manually analyze. For the transformations in the Partially Aligned category, we cluster the outputs based on the types of refactorings yielded by each refactoring detection tool. Then, we select one instance of each cluster to manually analyze. To make it simpler

Table 4: Summary of reported issues.

	Submitted	Duplicated	Not Bug	Open	Fixed
RefactoringMiner	20	2	3	0	15
RefDiff	14	1	0	12	1
Total	**34**	**3**	**3**	**12**	**16**

to explain to refactoring detection tool developers, we manually modify the program by removing the parts that are unrelated to the bug, inspired by delta debugging [25, 41]. Next, we manually analyze each candidate and discard the ones that we believe are not bugs. For example, if a tool reports refactoring `A` and it is not possible to transform, or even partially transform, the original code into the code after applying `A`, we consider the candidate as an issue. For the remaining ones, we report each pair of small input and output programs to refactoring detection tool developers stating that we expected the application of a single refactoring type.

By following this process, we submitted 34 issues. Table 4 summarizes them. Developers fixed 16 bugs, where RefactoringMiner fixed 15 bugs and RefDiff fixed 1 bug. Moreover, 3 issues are duplicated, 3 issues were not accepted, and 12 issues are still open. We run the technique in RefactoringMiner 2.1.0[4] and RefDiff 2.0[5] after bug fixing, and our technique does not detect new issues.

As mentioned before, our technique reveals differences between the refactoring mechanics of IDEs and refactoring detection tools. Therefore, when inconsistencies arise, it does not mean that the issue is on the refactoring detection tool side. Furthermore, when a refactoring detection tool reports multiple potential results, and we classify it as Partially Aligned, it does not always mean that they are wrong. Sometimes the refactoring mechanics implemented by the IDE may add some additional optional changes.

Understanding the root cause of a bug is not an easy task, given that authors are not the tool developers. In what follows, we discuss our results in light of some of the comments we received while submitting bugs to the developers [12] in RefactoringMiner and RefDiff. Listing 6 is related to the support to Generic Types in the Push Down Method refactoring added by RefactoringMiner's developers. When moving a method that returns Generic Types, it returns the actual type corresponding to the type in each subclass. As another example, Listing 5 may be caused because fan-in relationships (methods which call the refactored method) are ignored in the replacement function implemented by the matching algorithm in RefactoringMiner. A similar issue of that in Listing 6 was reported to RefDiff's developers. It detects the Move Method refactoring when we apply the Push Down Method refactoring. Developers explain that the tool enforces the same signature when searching Push Down Method candidates. Moreover, they do not deal with situations when generic types are replaced by concrete types. RefDiff may have classified it as the Move Method refactoring because this type allows changes to the signature. On the other hand, in Listing 7, RefDiff detects the Extract And Move Method refactoring, but we apply the Rename Method refactoring. Some code fragments are updated to a new method name. The renamed method is a single-line method, and this can increase the similarity score when comparing with updated code fragments.

[4]https://github.com/tsantalis/RefactoringMiner/commit/149468e
[5]https://github.com/aserg-ufmg/RefDiff/commit/3dabc79

Concerning the Extract Method refactoring, some of the additional refactorings reported by RefactoringMiner are due to the IDE refactoring mechanics and RefactoringMiner correctly reports them. For example, the Parameterize Variable refactoring is reported when local variables declared in the original method become parameters of the extracted method. Also, the Rename Parameter refactoring is reported when a parameter of the original method is passed with a different name in the extracted method.

We tried to contact RefDiff developers, but we did not receive an answer for 12 out of the 14 issues reported. In 6 of the 12 unanswered issues of RefDiff, RefactoringMiner is aligned with Eclipse. In 3 out of 12 unanswered issues, RefDiff and Eclipse are different, while RefactoringMiner is partially aligned with Eclipse.

4.4.2 Refactoring Detection Tools yield an empty set. In some cases, the refactoring detection tools do not detect the refactoring applied using Eclipse. For example, Listing 2 shows the application of the Move Method refactoring to the `doHealthCheck` method (Step 1). Eclipse moves the method to the `Health` class, changes its signature, and removes the original `doHealthCheck` method. RefactoringMiner does not detect this transformation. The Move Method refactoring mechanics described by Fowler [9] allows changing the signature of the method. In our study, we identify 8 issues related to this kind of refactoring. This example may help developers discuss the correct refactoring mechanics for the Move Method refactoring. One may argue that the correct refactoring mechanics is to enclose operations that change the signature of the moved method, such as Add parameter, Remove Parameter, Change Parameter Type involving the Source or Target class types. We reported this problem on issue #133 and developers fixed it. The fix intends to match the body of methods when there is an inner class in the refactoring relationship. The patch code checks if the removed parameter looks like having the same type as the target class, but also if the added parameter seems to have the same type as the source class.

Listing 5 shows the result of Eclipse applying the Rename Method refactoring to the `setAttribute` method. RefactoringMiner does not detect this refactoring. We analyzed the resulting code and identified that the transformation applied by Eclipse happens in a method containing a single line of code. The developers fixed issue #140 of Listing 5 in RefactoringMiner 2.1.0.

```
@@ class UMLModelASTReader
- variableDeclaration.setAttribute(true);
+ variableDeclaration.setAttr(true);
@@ class VariableDeclaration
- public void setAttribute(boolean isAttribute) {
+ public void setAttr(boolean isAttribute) {
```

Listing 5: Using the Eclipse Rename Method refactoring.

Furthermore, they mention that single-line methods are tricky to detect. In general, it is harder to match statements near to similar single-line methods. RefactoringMiner uses abstraction and argumentation to deal with changes taking place in code statements. It matches two versions of the same method if they have an identical signature, that is same name, parameters, return type, parent class, and body. Their algorithm matches the added and deleted code elements to find code elements with signature changes, but similar methods in the same class can be confusing.

In Listing 6, Eclipse applies the Push Down Method refactoring to the `awaitIgnoreError` method, which returns a generic type. The actual return type is replaced in each subclass. RefactoringMiner 2.0.3 yields an empty set. We submitted issue #137, and this problem is fixed in RefactoringMiner 2.1.0. The fix checks if the return type T in the superclass is replaced by the return type in each subclass. We also submit an issue to RefDiff developers. They explained that this behavior is a limitation of their implementation. Some language constructs, such as generic types and lambdas, are challenging for Java refactoring detection tools. Our technique can help to improve them by showing some examples that may expose new rules to be considered in refactoring detection tools.

```
@@ abstract class PendingResultBase
- @Override
- public final T awaitIgnoreError() { ... }
@@ class DistanceMatrixApiRequest
+ @Override
+ public final DistanceMatrix awaitIgnoreError() { ... }
```

Listing 6: Using the Eclipse Push Down Method refactoring.

4.4.3 Mechanics Are Different. In Listing 7, Eclipse applies the Rename Method refactoring to the `isConstructor` method. RefDiff yields an Extract and Move Method refactoring for each method call, in this case. This result shows that RefDiff does not yield the applied refactoring, but several others that were not applied. We report issue #19 to RefDiff's developers.

```
@@ class UMLOperation
- public boolean isConstructor() {
+ public boolean isConst() {
    return isConstructor;
  }
@@ class UMLModelDiff
private void checkForOperationMoves()
  ...
- else if(r.isConstructor() == a.isConstructor() ...) {
+ else if(r.isConst() == a.isConst() ...) {
```

Listing 7: Using the Eclipse Rename Method refactoring.

As another example, Eclipse applies the Extract Interface refactoring to the `StaticMapsRequest` class. RefactoringMiner 2.0.3 does not detect the Extract Interface refactoring. Moreover, it yields the Change Return Type (12) refactoring (see Listing 8). We verify the resulting code and the default parameter of Eclipse is to replace types where possible to the extracted interface type. This parameter needs attention because additional transformations applied by the IDE can directly impact the refactoring detection tools. Issue #146 of Listing 8 was fixed, and RefactoringMiner 2.1.0 correctly detects the applied refactoring. Before fixing, the detection of Extract Interface considers the same signature and return types. After, it ensures the same signature but ignores the changed types.

```
@@ class StaticMapsRequest
- public StaticMapsRequest center(LatLng location) {
+ public IStaticMapsRequest center(LatLng location) {
@@ interface IStaticMapsRequest
+ public interface IStaticMapsRequest {
+ ...
+ IStaticMapsRequest center(LatLng location);
```

Listing 8: Using the Eclipse Extract Interface refactoring.

Eclipse applies the Inline Method refactoring to the `locationBias` method (see Listing 9). RefactoringMiner detects four instances of the Extract Variable refactoring. The method is called in four different locations. This example may help developers to further discuss the temporary variables introduced by the Inline Method refactoring mechanics. Eclipse's engine introduces additional statements, causing variable renames or the extraction of temporary variables. Thus, RefactoringMiner correctly reports the Extract Variable refactoring. We report issue #121, and it was fixed in RefactoringMiner 2.1.0. The fix checks if the method call in the first statement is the expression, or sub-expression, of the method invocation in the second statement, and checks it from the second statement to the first statement. In addition, the Extract Variable refactoring can be avoided in this operation, because a single statement uses this variable.

```
@@ class FindPlaceFromTextRequest
- public ... locationBias(LocationBias lb) {
- return param("locationbias", lb);
- }
@@ class PlacesApiTest {
+ LocationBias lb = new LocationBiasIP();
   ...
   .fields(...)
- .locationBias(new LocationBiasIP())
+ .param("locationbias", lb)
```

Listing 9: Using the Eclipse Inline Method refactoring.

4.4.4 Mechanics Are Partially Aligned. In Listing 10, Eclipse applies the Rename Method refactoring to `getNonMappedLeavesT1` method. RefactoringMiner 2.0.3 yields that the Change Variable Type refactoring is applied to the `statement` variable, but there is no change in this variable. We report issue #139, and developers fixed it in RefactoringMiner 2.1.0.

```
@@ class UMLOperationBodyMapper
- public ... getNonMappedLeavesT1() {
+ public ... getNonMLeavesT1() {
     return nonMappedLeavesT1;
@@ class InlineOperationRefactoring
- for(StatementObject s : b.getNonMappedLeavesT1()) {
+ for(StatementObject s : b.getNonMLeavesT1()) {
  ...
  for(CompositeStatementObject s :
       ↪ b.getNonMappedInnerNodesT1()) {
```

Listing 10: Using the Eclipse Rename Method refactoring.

Listing 11 shows that Eclipse applies the Rename Class refactoring to the `Replacement` class. RefactoringMiner detects the Rename Class refactoring along with other 245 refactoring instances, such as the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings, since the renamed class are used in several parts of the program. RefDiff has a similar behavior, and yields 18 instances of the Change Signature Method refactoring.

The other refactoring types reported, such as the Change Variable/Attribute/Return/Parameter Type refactorings, are updates to the places where the renamed type is referenced in variable types, return types, parameter types, and field types. Fowler [9] states that changing each use to the new class name is a step in the refactoring mechanics of the Rename Class refactoring. RefactoringMiner yields a coarse-grained refactoring, such as the Rename Class refactoring, and a number of fine-grained transformations used to derive the coarse-grained refactoring, such as, the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings. RefactoringMiner developers reply to a related issue (#120) explaining that they prefer to show multiple instances. The refactoring community should discuss about the granularity of each refactoring, and how they relate to coarse-grained transformations. This example may help developers to discuss the correct refactoring mechanics for the Rename Class refactoring. One may argue that the correct refactoring mechanics is to exclude the instances of Change Variable/Attribute/Return/Parameter Type refactoring for which the type change corresponds to the Renamed Class.

```
@@ class Replacement;
- public class Replacement {
+ public class Replace {
@@ class AbstractCodeMapping
  private boolean contains(String v) {
- for(Replacement r : getReplace()) {
+ for(Replace r : getReplace()) {
@@ class TernaryOperatorExpression
- public Replacement m(String s) {
+ public Replace m(String s) {
```

Listing 11: Using the Eclipse Rename Class refactoring.

In Listing 12, Eclipse applies the Move Method refactoring to the `consistencyCheck` method. RefactoringMiner yields the Inline Method and the Extract and Move Method refactorings. However, Eclipse does not apply the Inline Method instances. We report issue #143, and developers fixed it.

```
@@ class VariableDeclaration
+ boolean consistencyCheck(...) {
@@ class VariableReplacementAnalysis
- consistencyCheck(v1, v2, set);
+ v1.consistencyCheck(this, v2, set);
  ...
- private boolean consistencyCheck(...) {
```

Listing 12: Using the Eclipse Move Method refactoring.

Suppose a developer pushed down a method to `N` subclasses. RefactoringMiner and RefDiff yield `N` instances of the Push Down Method refactoring. For example, Listing 13 shows that Eclipse applies the Push Down Method refactoring to the `normalizedEditDistance` method, moving it to 11 subclasses using Eclipse. RefactoringMiner yields 11 instances of the Push Down Method refactoring. RefDiff also yields the same output. According to Fowler's Push Down Method mechanics [9], this should be considered a single instance of the Push Down Method refactoring.

This example may help developers to discuss the correct refactoring mechanics for the Push Down Method, Extract Method and Inline Method refactorings. Eclipse's default refactoring parameter is to copy/extract the same code fragments. This parameter can fix design flaws, such as Duplicated Code [9]. However, the end user can change this behavior in GUI and the refactoring mechanics may be different when a user selects different parameters. One may argue that the correct refactoring mechanics is to exclude the extra refactoring instances reported due to the mechanics. We reported issues #120 and #122 to RefactoringMiner's developers, but they did not accept the issues.

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

Listing 13: Multiple instances of the Push Down Method refactoring in RefactoringMiner and RefDiff.

4.5 Threats to Validity

In this section, we discuss some threats to validity. We do not evaluate real transformations applied by developers. However, we apply a number of transformations to four real open-source projects using a popular IDE (Eclipse). In addition, we manually analyze the candidates yielded after Step 3 of our technique. Since this manual classification is a time consuming and error-prone activity, we may miss some bugs. We submit 34 issues to the developers of the refactoring detection tools. They fixed 16 bugs, 3 were not accepted, and 12 issues are still open. During our manual analysis, we did not find any bugs in Eclipse, even though they might be present, as reported in previous works [16, 23].

In our study, we only evaluated open source projects hosted in GitHub. However, the evaluated projects have been actively developed for more than six years. We analyze eight types of refactoring, such as the Rename refactoring, which is frequently applied by developers [20]. Table 3 presents the number of issues reported to the Rename Method and Rename Class refactorings. Moreover, refactoring implementations may introduce behavioral changes when performing a refactoring [33]. As future work, we intend to improve our technique by using SafeRefactor [35] after Step 2 to discard transformations that introduce behavioral changes.

The default parameters used in our experiment are subject to human errors. However, we addressed that point by inspecting the source code after the refactoring when we intend to submit the issues. For example, we specify the parameter of Rename Method refactoring as the method's same with a suffix. In future work, we intend to verify new values for these parameters. We only evaluated the refactoring implementations of Eclipse, for a popular programming language (Java). Eclipse is a popular IDE and is used by developers to apply refactorings [22]. We also evaluate the best refactoring detection tools available [31, 37].

5 RELATED WORK

Silva et al. [31] propose a language-agnostic refactoring detection tool RefDiff 2.0. It presents a new refactoring detection algorithm that abstracts from specific programming languages through Code

Structure Trees. This abstraction allows supporting different languages, such as Java, C, and JavaScript. Tsantalis et al. [37] propose RefactoringMiner 2.0, a refactoring detection tool for Java. It relies on an AST-based statement matching algorithm that determines refactoring candidates without requiring user-defined thresholds and covers 40 refactoring types, 25 more than the previous version.

Tsantalis et al. [37] execute RefactoringMiner 2.0, GumTreeDiff and two versions of RefDiff on all 536 commits from 185 open-source GitHub-hosted projects monitored over a two-months period on an existing dataset [38] and considered the union of all true positives as the ground truth. Two authors validated the refactoring instances [37]. It includes 7,226 true positives in total, for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools. RefDiff initially used this dataset [38] to evaluate precision and recall. They also manually included other instances. In our work, we propose a technique to automatically test refactoring detection tools. We evaluate eight refactoring types using 9,885 transformations applied by Eclipse to evaluate two refactoring detection tools. Our work can help refactoring detection tool developers to improve their dataset, and find some transformations that may help them improving the refactoring detection rules, and also to improve refactoring implementations.

Prete et al. [26] develop Ref-Finder, which detects refactorings using a template-based refactoring reconstruction approach. Their tool can identify 63 of 72 refactoring types from Fowler's catalog [9]. To evaluate their tool, they performed two case studies: they create code samples from Fowler's catalog, and they select version pairs from open-source projects. Ref-Finder achieved an overall precision and recall of 79% and 95%, respectively. Dig et al. [5] present an algorithm that detects refactorings performed during component evolution. Their algorithm was implemented as an Eclipse plugin called RefactoringCrawler. They evaluate their tool in three components ranging with 17 KLOC up to 352 KLOC, and its accuracy was over 85% for seven types of refactorings. Out technique may be used to test their refactoring detection tools.

Oliveira et al. [23] propose a technique to identify differences in refactoring mechanics used by tool developers of refactoring implementations. They perform a pairwise comparison of 10 types of refactorings to 157,339 programs using 27 refactoring implementations from Eclipse, JRRT, and NetBeans. Oliveira et al. [22] also conduct a survey with 107 developers of popular Java projects to better understand the refactoring mechanics used by them in practice. They found that most developers expect the refactoring output based on their experience and there is no consensus 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. In our work, we apply eight types of refactorings to four real open-source projects, and compare the differences between mechanics of two refactoring detection tools and Eclipse IDE. The differences found motivate the importance of discussing refactoring mechanics by our community.

Soares et al. [34] compare three different approaches based on manual analysis, commit message and dynamic analysis using SafeRefactor [35] to detect refactorings considering behavioral preservation and found the Ref-Finder presented a low precision and recall. Mongiovi et al. [17, 18] improve SafeRefactor by including change impact analysis and skips. Similar bugs occur in other domains [7, 13, 14]. We intend to use SafeRefactor after Step 2 to only consider behavior-preserving transformations.

Gligoric et al. [10] automatically apply a number of refactorings to identify bugs in refactoring implementations. They find a number of bugs related to compilation errors. We also propose a tool to automatically apply refactorings in all possible locations in a program. However, our goal is to identify differences in the refactoring mechanics of Eclipse and refactoring detection tools.

There are several works that find the sets of statements to be extracted. Tsantalis and Chatzigeorgiou [36] propose an approach to select related statements that can be extracted. They consider two aspects to identify related statements. First, it selects all statements that computes a given variable. Second, it extracts the statements affecting the state of a given object. Their approach allows producing meaningful and behavior preserving refactoring opportunities. Silva et al. [32] propose a rank function to classify initial candidates according to their potential to improve program comprehension. Their approach tends to encapsulate well-defined computation with its own set of dependencies and that is also independent of remaining statements of the original method. Charalampidou et al. [3] suggest resolving the Long Method smell by using the Single Responsibility Principle to identify opportunities of Extract Method refactoring. The approach calculates cohesion between pairs of statements to determine code fragments that collaborate for functionality. Moreover, the approach identifies statements that perform the same functionality. Xu et al. [40] propose a machine-learning-based approach to recommend Extract Method refactorings based on complexity, cohesion and coupling. They use samples, which were obtained from real-world Extract Method refactorings, to train the probabilistic model. Their tool was evaluated on five open-source repositories and compared against state-of-art approaches: SEMI [3], JExtract [32] and JDeodorant [8]. In our work, first we try to extract the second statement. If we cannot apply a refactoring, we try to extract the second and third statements. We repeat this process by adding more statements until we successfully apply a refactoring using the IDE or we reach the last statement.

6 CONCLUSION

We propose a technique to test refactoring detection tools. We evaluate 9,885 transformations applied to four real open-source projects using eight refactoring types of Eclipse. RefactoringMiner and RefDiff are aligned with the refactoring mechanics of Eclipse in 74.28% and 78.45% of the transformations, respectively. We report 34 issues to the developers of RefactoringMiner and RefDiff. They fixed 16 bugs, 12 bug reports are still open, 3 bugs are duplicated, and 3 issues are not accepted.

Our results may be useful for developers of refactoring detection tools and refactoring implementations to discuss about what should be considered in the refactoring mechanics of each refactoring type. This process may help to improve both tools. Moreover, our technique may be useful to improve the process of creating a better dataset to be used to evaluate refactoring detection tools, since it can automatically yield a number of transformations. Furthermore, it avoids the bias of the authors of the refactoring detection tool to manually classify transformations [37]. To minimize the problem of depending on the refactoring mechanics implemented by IDEs,

developers may use advanced refactoring tools, such as Refazer [27, 28]. The tool is able to generate a transformation based on few examples of transformations given by the user.

As future work, we intend to evaluate more types of refactorings and increase the number of evaluated projects. We aim at improving our oracle since the current version requires some manual steps and rely on Eclipse's implementations. We also intend to evaluate composite refactorings [1, 2] by applying a sequence of refactorings using Eclipse and use refactoring detection tools to identify them. We also aim at evaluating refactorings implemented by other popular IDEs, such as IntelliJ. We also intend to use our technique in other programming languages, such as C or JavaScript. RefDiff can detect refactorings applied to them. We also aim at improving our technique to focus on applying refactorings only on realistic opportunities. Furthermore, we will consider using the previous approaches [3, 32, 36, 40] to find the sets of statements to be extracted in the Extract Method refactoring.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions. This work was partially supported by CNPq and CAPES grants.

REFERENCES

[1] Ana Bibiano, Wesley Assunção, Daniel Coutinho, Kleber Santos, Vinícius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, Caio Barbosa, João Marques, and Anderson Oliveira. 2021. Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 298–308.

[2] Ana Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *Proceedings of the Int. Conference on Program Comprehension (ICPC)*. 149–159.

[3] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *TSE* 43, 10 (2017), 954–974.

[4] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *ESEC/FSE*. ACM, 185–194.

[5] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 404–428.

[6] Eclipse.org. 2022. Eclipse Project. http://www.eclipse.org.

[7] Leonardo Fernandes, Márcio Ribeiro, Luiz Carvalho, Rohit Gheyi, Melina Mongiovi, André Santos, Ana Cavalcanti, Fabiano Cutigi Ferrari, and José Carlos Maldonado. 2017. Avoiding useless mutants. In *Proceedings of the Generative Programming: Concepts and Experiences (GPCE)*. 187–198.

[8] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Identification and Application of Extract Class Refactorings. In *ICSE*. 1037–1039.

[9] Martin Fowler. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley.

[10] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. 2013. Systematic Testing of Refactoring Engines on Real Software Projects. In *ECOOP*. 629–653.

[11] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *FSE*. 50:1–50:11.

[12] Osmar Leandro, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, and Alessandro Garcia. 2022. A Technique to Test Refactoring Detection Tools (artifacts). https://github.com/osmarleandro/comparing-mechanics.

[13] 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.

[14] Flávio Medeiros, Iran Rodrigues, Márcio Ribeiro, Leopoldo Teixeira, and Rohit Gheyi. 2015. An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *Proceedings of the Generative Programming: Concepts and Experiences (GPCE)*. 35–44.

[15] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

[16] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering* 44, 5 (2018), 429–452.

[17] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making refactoring safer through impact analysis. *Science of Computer Programming* 93 (2014), 39–64.

[18] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. 2014. Scaling Testing of Refactoring Engines. In *Proceedings of the Int. Conference on Software Maintenance and Evolution (ICSME)*. 371–380.

[19] Emerson Murphy-Hill, Moin Ayazifar, and Andrew P. Black. 2011. Restructuring software with gestures. In *VL/HCC*. 165–172.

[20] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *TSE* 38, 1 (2012), 5–18.

[21] Stas Negara, Nicholas Chen, M. Vakilian, Ralph Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP*. 552–576.

[22] 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.

[23] Jonhnanthan Oliveira, Rohit Gheyi, Felipe Pontes, Melina Mongiovi, Márcio Ribeiro, and Alessandro Garcia. 2020. Revisiting Refactoring Mechanics from Tool Developers' Perspective. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)*. 25–42.

[24] William Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[25] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java reflection API: revealing the dark side of the mirror. In *Proceedings of the Foundations of Software Engineering (FSE)*. 636–646.

[26] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based Reconstruction of Complex Refactorings. In *ICSM*. 1–10.

[27] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 404–415.

[28] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. 74–83.

[29] Max Schäfer and Oege de Moor. 2010. Specifying and implementing refactorings. In *OOPSLA*. 286–301.

[30] Danilo Silva. 2020. *Mining Refactorings from version histories: studies, tools, and applications.* Ph.D. Dissertation. Federal University of Minas Gerais.

[31] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802.

[32] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending Automated Extract Method Refactorings. In *ICPC*. 146–156.

[33] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

[34] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. 2013. Comparing Approaches to Analyze Refactoring Activity on Software Repositories. *Journal of Systems and Software* 86, 4 (2013), 1006–1022.

[35] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making Program Refactoring Safer. *IEEE Software* 27, 4 (2010), 52–57.

[36] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

[37] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930 – 950.

[38] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *ICSE*. 483–494.

[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 *ICSE*. 233–243.

[40] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. GEMS: An Extract Method Refactoring Recommender. In *ISSRE*. 24–34.

[41] Andreas Zeller. 2009. *Why Programs Fail: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers.