Automating Test-Specific Refactoring Mining: A Mixed-Method Investigation

Luana Martins*, Heitor Costa†, Márcio Ribeiro‡, Fabio Palomba§, Ivan Machado*
martins.luana@ufba.br, heitor@ufla.br, marcio@ic.ufal.br, fpalomba@unisa.it, ivan.machado@ufba.br
*Federal University of Bahia, Salvador, Brazil
†Federal University of Lavras, Lavras, Brazil
‡Federal University of Alagoas, Maceió, Brazil
§Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy

***Abstract*—Refactoring is a practice commonly used by developers to restructure the source code without changing its external behavior. Over the last decades, the software engineering research community has been making use of mining software repository techniques to investigate refactoring under multiple perspectives, identifying properties and impact of this practice on source code quality, other than using refactoring data coming from software repositories to build automated recommendation systems. While the current state of the art proposes various automated tools to mine refactoring data, there is still a lack of instruments that may help researchers when mining test-specific refactoring data. The availability of those instruments may enable additional, specialized techniques to support developers while refactoring test code. In this paper, we introduce an approach that extends REFACTORINGMINER—a well-established refactoring mining tool having high precision and recall scores—and is able to detect seven test-specific refactoring operations. We perform mixed-method research to assess capabilities and usefulness of the approach. First, we compare the test-specific refactoring data extracted by the approach against an oracle of 375 test-specific refactorings. Second, we engage with 15 software engineering researchers and apply a technology acceptance model to investigate how they would benefit from our approach. The key results of the study show that our approach reaches 100% and 92.5% of precision and recall scores, respectively. In addition, the approach is considered useful and suitable for various research tasks, including the definition of novel learning models able to recommend test-specific refactoring actions.**

***Index Terms*—Software testing; Test-Specific refactoring, Refactoring Mining; Mining Software Repositories.**

I. INTRODUCTION

Refactoring is a key practice that provides developers with a means to improve software quality without changing its external behavior [1]. Since decades, refactoring has been attracting the interest of the software engineering research community, which investigated the matter under multiple perspectives, uncovering the reasons why developers perform refactoring [2]–[4], the concerns that prevent its wider application [5]–[7], and benefits and drawbacks for software maintenance and evolution [8], [9]. For instance, researchers showed that refactoring may enable the removal of code smells [10], [11] and self-admitted technical debt [12], other than the improvement of program comprehension [13] and language constructs adaptation [14]. At the same time, researchers have been identifying cases where the application of refactoring may induce the introduction of defects [15], design concerns [16], and vulnerabilities [17], and cause changes to clients' libraries and frameworks [18].

Such an extensive body of knowledge has been mostly enabled by the availability of open data coming from public software repositories (e.g., GITHUB), which provided researchers with the capability of mining refactoring data to learn properties and practices, inform the definition of automated refactoring recommendation systems, and investigate the impact of refactoring on source code quality [19], [20]. In this respect, the amount of refactoring mining instruments proposed in literature [21]–[24] has facilitated the data collection and analysis procedures applied by researchers.

In such a context, we point out that most effort has been devoted to the definition of automated approaches for mining refactoring data, while there is still a lack of instruments that could facilitate researchers in the task of mining *test-specific refactoring actions* performed by developers over software maintenance and evolution. Detecting refactoring from the test code perspective would have a notable impact on the capabilities of researchers to support test code evolution and refactoring. Indeed, despite many previous authors have raised the need for more research on test code refactoring [25]–[27], the current knowledge and support is limited [9], hence calling for novel enabling approaches.

In this paper, we propose a tool-supported approach able to detect test-specific refactoring operations. Our approach is built on top of REFACTORINGMINER [24], [28], one the most established refactoring mining instruments: We first integrate the test detection mechanisms proposed by REFACTORINGMINER team in late 2021[1]; secondly, we extended the set of changes reported by REFACTORINGMINER through the detection of additions, removals, and modifications on test assertions; finally, we used the set of changes to implement rules for detecting seven test refactoring operations.

We assess the soundness and usefulness of our approach through mixed-method research [29]. First, we compute the precision and recall of the approach by comparing its output against a manually-validated dataset of test-specific refactoring actions, which we collect and curate from 13 open-

[1]Available at https://github.com/tsantalis/RefactoringMiner/pull/225

source projects pertaining to the Apache Software Foundation. Secondly, we involve 15 software engineering researchers experienced in software testing and software maintenance and evolution within the scope of a technology acceptance model construction [30], which provides insights into the perceived usefulness and ease of use of the devised approach.

On the one hand, the results indicate that our approach detects the seven test-specific refactoring operations with an average precision and recall of 100% and 92.5%, respectively. On the other hand, the technology acceptance model let arise the tasks for which our approach may be used in the future and the new challenges that research might address with it.

To sum up, our work brings the following contributions:

1) A tool-supported approach for mining seven test-specific refactorings, which researchers may use to enlarge the body of knowledge on test-specific refactoring;
2) A novel, curated dataset of test-specific refactoring data, which researchers may use to understand further how test-specific refactoring is performed in practice;
3) The results coming from the technology acceptance model construction, which point out future research challenges that our approach can help address;
4) An online appendix [31] that contains all data/scripts used in this study and that researchers may use to either reproduce our work or build on top of that.

II. Related Work

A. Overview of the Refactoring Mining Tools

Early researchers proposed detection approaches and algorithms relevant to establishing a theoretical foundation for refactoring detection. In order to understand how and why the systems evolve, Demeyer et al. [32] proposed four heuristics for identifying refactorings using reverse engineering of reconstructed code. They performed case studies on different versions of three systems and concluded that these heuristics help to reveal where, how, and why an implementation has drifted from its original design.

Weissgerber and Diehl [33] proposed a technique to detect refactoring candidates by finding pairs of code elements (i.e., classes, methods, fields) with some differences in their signatures. Then, the technique ranks the refactoring candidates based on the results of a code-clone detection algorithm implemented through the CCFinder (Code Clone Finder) tool [34]. The CCFinder tool implements a technique that transforms source code into text and performs a token-by-token comparison. In addition, Weissgerber and Diehl [33] created an oracle by inspecting commit messages for references to refactoring operations, which they used to evaluate the accuracy of the proposed technique.

Later on, other developers used such theoretical foundations to develop refactoring detection tools. Dig et al. [21] developed the first comprehensive detection tool called RefactoringCrawler. The tool performs a syntactic analysis to find similar fragments in text files using an Information Retrieval-based algorithm. Then, the tool performs a semantic analysis based on the reference graphs to determine whether the fragments represent a refactoring. The tool is an *Eclipse* plug-in and detects seven types of refactorings in Java code, focusing on rename and move refactorings.

Xing et al. [35] developed JDEvAn (Java Design Evolution Analysis), a tool that automatically detects the design changes between two versions of a system. In more detail, JDEvAn implements a Java fact extractor to reverse engineer logical design models from source code. Then, it implements the *UMLDiff* algorithm [36] to detect design changes, which feed a set of queries for the refactorings detection. The authors performed three case studies using the JDEvAn tool, showing its effectiveness in practice.

Prete et al. [37] developed Ref-Finder, an Eclipse plug-in that takes two program versions as input and detects 63 refactorings from the Fowler's catalog [1]. The tool is based on the tool LSdiff (Logical Structural Diff) which computes the delta between two versions of the source code and detects refactorings using a template-based refactoring reconstruction approach. For the evaluation, the authors run the tool against extracted code examples from Fowler's book and released pairs of jEdit, showing the tool's capabilities.

Silva and Valente [38] developed RefDiff, a tool that combines heuristics based on static analysis and code similarity to detect 13 refactorings. The authors evaluated the tool's accuracy using an oracle of seeded refactorings performed by graduate students. Afterward, they used the same approach to develop the first multilanguage refactoring detection tool, called RefDiff 2.0. The authors evaluated the tool's accuracy in detecting refactorings in code written in Java, JavaScript, and C programming languages. Regarding the tool's accuracy on Java projects, they relied on an available oracle comprising 536 commits from 185 open-source projects [28].

Tsantalis et al. developed the RMiner (version 1.0) [28] and RefactoringMiner (version 2.0) [24], that implement an AST-based statement matching algorithm and applies a set of rules for detecting code refactorings. In its first version, RMiner detects 15 high-level refactorings from Fowler's catalog [1]. The authors created a dataset with 3,188 real refactorings instances from 185 open-source projects to evaluate the RMiner. Differently, RefactoringMiner (version 2.0) builds upon its predecessor to support the detection of 40 low-level refactorings. In addition, the authors extend their oracle to comprise 7,226 true instances of refactorings.

B. Extensions of RefactoringMiner tool

Several studies adapted RefactoringMiner to detect code refactorings in different programming languages. Kurbatova from JetBrains Research leads the development of kotlinRMiner, a library that extends RefactoringMiner tool to detect 19 code refactorings in Kotlin code. Later on, Kurbatova et al. [39] presented RefactoringInsight, a plugin for IntelliJ IDEA built on top of the kotlinRMiner library and RefactoringMiner tool to detect 19 code refactorings in Kotlin and 40 Java code refactorings. In addition, that extended tool provides developers with different views to

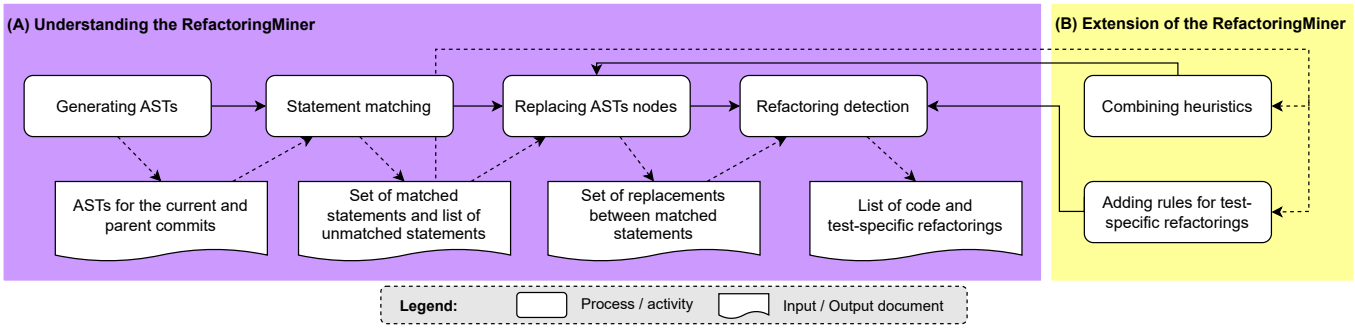

Fig. 1: Understanding the current state and extension of *RefactoringMiner* tool.

show a list of code refactorings per commit or pull request and the history of refactorings for methods and classes. Lastly, the authors analyzed the time spent detecting the code refactorings into 960 commits of four projects.

Atwi et al. [40] focused on Python projects, presenting the PYREF tool to detect four method-level refactorings. The authors evaluated that tool precision against a manually built oracle with 327 true and false positive instances. As a result, the tool scored 89.60% and 76.10% for precision and recall. Similarly, Dilhara et al. [41] presented the R-CPATMINER tool that detects 18 refactorings in Python code. The authors adapted it to create a graph-based representation of Python projects, combining it with the PY-REFACTORINGMINER tool, which detects code refactorings in Python code. The tool achieves an average precision of 97.34%. Later, Dilhara et al. [41], [42] used the tool to mine 4,166,520 commits from 1,000 Machine Learning application systems to understand the challenges and frequent code change patterns performed by developers while evolving those systems.

Inspired by REFACTORINGMINER tool, Shiblu [43] presented the JSDIFFER tool, which supports the detection of 18 code refactorings in JavaScript projects. As the structure of JavaScript code differs significantly from Java code, the author made several adaptations in terms of structural mapping. Additionally, the author built an oracle of 341 code refactoring instances mined from 18 open-source JavaScript projects. Lastly, the authors evaluated the tool's accuracy, scoring 97% and 45% for precision and recall, respectively.

Differently, Kim et al. [44] extended REFACTORINGMINER tool to detect 12 refactorings in test code annotations. The authors evaluated the tool accuracy against an oracle containing 638 annotation change instances, scoring 99.7% and 98.7% for precision and recall, respectively. In addition, the authors studied the developers' maintenance activities on test annotations. They created a taxonomy by manually inspecting and classifying a sample of test annotation changes and documenting the motivations driving those changes.

Looking at the official repository of REFACTORINGMINER,[2] it is possible to observe that some attempts to cope with test refactoring operations have been proposed in late 2021. In particular, one of the repository contributors opened a pull request[3] that detects the migration of expected exception features between JUnit versions. Nonetheless, those preliminary attempts have not yet been further explored in the literature—our paper makes a first step toward this direction.

Unlike the papers discussed above, our extension of REFACTORINGMINER aims to detect test-specific refactoring operations applied by developers over software evolution. While the previous versions of REFACTORINGMINER tool detects refactorings from Fowler's catalog [1], migrations from JUnit versions,[3] and changes in methods' annotations [44] of both production and test code, we focused on general test design and assertions. Therefore, our extension builds on top of REFACTORINGMINER tool and other extensions by introducing new rules for mining refactoring actions in the test code.

III. INTRODUCING TESTREFACTORINGMINER

Fig. 1 presents an overview of the current state of REFACTORINGMINER tool and its extension to detect seven test-specific refactorings. We selected the test-specific refactorings based on two criteria. First, the refactorings represent solutions to fix the most popular test smells in open-source projects [45], [46]. Second, we manually identified the most popular refactorings in the Apache Foundation ecosystem to understand how developers apply the test code changes.

In the following, we first showcase the main characteristics of REFACTORINGMINER and, afterwards, we present our approach to detect test-specific refactorings.

A. Understanding RefactoringMiner

We chose to extend the state-of-the-art refactoring mining tool (REFACTORINGMINER [24], [28]) with detection rules for test-specific refactorings for the following reasons:

- The tool has the highest precision (99.8%) and recall (97.6%) scores among the currently available refactoring mining tools, hence allowing us to build on top to the current state of the art;
- The tool encapsulates the changes between the matched and unmatched statements (i.e., new/deleted statements) within an object that provides information for detecting the new test-specific refactorings;

[2]Available at https://github.com/tsantalis/RefactoringMiner

[3]Available at https://github.com/tsantalis/RefactoringMiner/pull/225

- The tool only analyzes added, deleted, and changed files, reducing the time and computational resources for detecting refactorings in large projects.

Specifically, REFACTORINGMINER differs from other refactoring mining tools (e.g., REF-FINDER [37]) by analyzing only the added, deleted, and changed files between two project versions. For each version, the tool creates an AST for those files without resolving binding information from the compiler (Fig. 1A - *Generating ASTs*). Then, it implements a three-step algorithm to match the statements between two code fragments following a bottom-up approach. First, the algorithm matches the statements with identical string representations and nesting depth. Second, the algorithm matches the statements with identical string representations regardless of their nesting depth. In the third step, the algorithm matches the statements that become identical after replacing the AST nodes being different between the two statements (Fig. 1A - *Statement matching*).

The tool implements an algorithm that receives two statements and performs replacements of AST nodes until the statements become textually identical (Fig. 1A - *Replacing ASTs nodes*). The algorithm excludes from the replacements list the AST nodes common in both statements (e.g., variable identifiers, types, literals, and operators) and the AST nodes that cover the entire statement (e.g., an expression statement that is a method invocation or class instance creation followed by the operator ';') to reduce the excessive number of matching statement combinations. Next, the algorithm calculates and combines 13 heuristics to allow the matching of textually different AST nodes that cover the entire statement. They are: (1) identical receiver expression, (2) identical method invocation name, (3) identical list of arguments, (4) argument added/deleted, (5) argument split/merged, (6) argument wrapped, (7) argument replaced, (8) renamed method invocation, (9) different receiver expression, (10) field assignment replaced with setter invocation, (11) class instance creation replaced with builder call chain, (12) method invocation replaced with conditional expression, and (13) split/merge invocation to/from multiple statements.

Finally, the tool implements rules for detecting a set of 93 code refactorings for six different types of code elements, i.e., packages, type declarations, methods, fields, local variables/parameters, and type references (Fig. 1A - *Refactoring detection*). As a result, the tool reports a set of replacements between matched statements and a list of unmatched statements (i.e., new/deleted statements).

B. Extending RefactoringMiner for test code

Developers often rely on specific constructs of testing frameworks to develop test code. For example, developers invoke the assert methods from the `Assert` class of the JUNIT framework to assert the expected outcomes. When the two statements under analysis (parent and current commits) are assert method invocations, the replacement corresponds to AST nodes that cover the entire statement (Fig. 1A - *Replacing ASTs nodes*). Therefore, the current version of REFACTORINGMINER tool excludes those assert method invocations from the replacements list and does not analyze them to detect code refactorings.

To overcome this limitation, we extended the tool to get replacements of assert methods combining the four heuristics prior cited [24]: (i) Identical receiver expression, (ii) Identical method invocation name, (iii) Argument added/deleted, and (iv) Renamed method invocation (Section III-A). We combined the heuristics (i), (ii), and (iii) to identify added/deleted arguments of method invocation names identical and the heuristics (i), (ii), and (iv) heuristics to identify added/deleted arguments of names of method invocation different (Fig. 1B - *Combining heuristics*). The combination of heuristics results in a set of replacements between matched statements, which we used to create detection rules for the test-specific refactorings (Fig. 1B - *Adding rules for test-specific refactorings*):

(a) Add explanation message. Developers should use the optional parameter of the JUNIT assert methods to provide an explanatory message to the user when the assertion fails. The assert methods receive a string message in the first parameter for JUNIT4 or older versions, and in the last parameter for newer JUNIT versions.

(b) Replace reserved words. Instead of passing reserved words as parameters for the assert methods, developers should use the appropriate assert methods from the testing framework. For example, developers should use `assertNull` to verify whether the value of one object is `null`, `assertTrue` to verify whether the value of one object is `true`, or `assertFalse` to verify whether the value of one object is `false`.

(c) Split conditional parameters. Developers should not force conditional expressions into the parameters of the `assertTrue` and `assertFalse` methods to verify whether two objects are equal, or whether one object contains another. Instead, developers should pass the objects as parameters for the `assertEquals` or `assertThat` methods, respectively.

(d) Replace the not (!) operator. For readability purposes, the assert methods are given in pairs to assert whether a condition is true (`assertTrue`) or false (`assertFalse`), two objects are equal (`assertEquals`) or different (`assertNotEquals`), two objects refer to the same object (`assertSame`) or different objects. Instead of creating a conditional logic in the assertions using the `not (!)` operator, developers should use an equivalent assert method.

In addition, we used the list of unmatched statements of the test code to derive new test-specific refactorings. For example, developers insert logic in the test methods when using try/catch statements for handling exceptions with JUNIT3. JUNIT4 solved this problem by introducing the `@Rule` and `@Test expected` annotations, and JUNIT5 introduced the `assertThrows` method to fail the test methods. To detect whether developers migrated from JUNIT3 to JUNIT5 or from JUNIT4 to JUNIT5, we analyzed the unmatched statements, i.e., the deleted leaf statements from the parent commit and the

Refactoring	Parent Commit	Current commit
(a) Add explanation message	1 assertTrue(boolean condition); 2 assertEquals(Object expected, Object actual)	assertTrue(String message, boolean condition); assertEquals(String message, Object expected, Object actual);

(b) Replace reserved words	1 assertEquals(String message, boolean condition, ReservedWord true);
	2 assertEquals(String message, boolean condition, ReservedWord false);
	3 assertEquals(String message,boolean condition, ReservedWord null);
		assertTrue(String message, boolean condition);
		assertFalse(String message, boolean condition);
		assertNull(String message, boolean condition);

(c) Split conditional parameters	1 assertTrue(String message, boolean expected.equals(actual));
	2 assertTrue(String message, boolean expected == actual);
	3 assertTrue(String message, boolean expected.contains(actual));
		assertEquals(String message, Object expected, Object actual);
		assertEquals(String message, Object expected, Object actual);
		assertThat(String message, T actual, Matcher<T> matcher);

(d) Replace the not (!) operator	1 assertTrue(String message, !boolean condition);
	2 assertNull(String message, !boolean condition);
	3 assertEquals(String message, !Object expected, Object actual);
		assertFalse(String message, boolean condition);
		assertNotNull(String message, boolean condition);
		assertNotEquals(String message, Object expected, Object actual);

(e) Replace try/catch with asserThrows	1 try {
	2 statements;
	3 fail(String failMessage);
	4 } catch (ExpectedException exception) {
	5 statements;
	6 }
		assertThrows(ExpectedException.class,
		() -> {statements}, String failMessage);

(f) Replace @Rule annotation with assertThrows	1 @Rule
	2 public ExpectedException exception = ExpectedException.none();
	3
	4 @Test
	5 public void testMethod() throws ExpectedException {
	6 exception.expect(ExpectedException.class);
	7 statements;
	8 }
		@Test
		public void testMethod() {
		assertThrows(ExpectedException.class,
		() -> {statements}, String message);
		}

(g) Replace @Test annotation with assertThrows	1 @Test(expected = ExpectedException.class)
	2 public void testMethod() throws ExpectedException {
	3 statements;
	4
	5 }
		@Test
		public void testMethod() {
		assertThrows(ExpectedException.class,
		() -> {statements}, String message);
		}

Fig. 2: Extension of rules for detecting the test-specific refactorings. The column parent commit highlights in red the removal of code, and the current commit highlights in green the addition of code.

newly added leaf statements from the current commit. From a technical perspective, we exploited the source code developed by one of the contributors of the repository in late 2021 [47] to add the detection rules for three test-specific refactorings (Fig. 1B - *Adding rules for test-specific refactorings*):

(e) Replace try/catch with assertThrows. Instead of using `try/catch` blocks to insert logic for handling exceptions, developers should use the `assertThrows` method of JUNIT5.

(f) Replace @Rule annotation with assertThrows. JUNIT4 uses the `@Rule` annotation to check whether a method throws an exception. That annotation allows developers to write the test methods without assertion statements. Therefore, they replace the `@Rule` annotation with the `assertThrows` method.

(g) Replace @Test annotation with assertThrows. The `@Text expected` annotation of JUNIT4 indicates an exception can be thrown anywhere in the test method, not requiring developers to write the assertions. Therefore, developers replace the `@Test expected` annotation with the `assertThrows` method.

Fig. 2 presents excerpts of test code in JUNIT4 to exemplify the test-specific refactorings supported by our TESTREFACTORINGMINER[4]. The column *parent commit* highlights the removal of code in red, and the column *current commit* highlights the addition of code in green. It is worth noticing that the test-specific refactorings from (a) to (d) are independent from the JUNIT version. For example, to detect the *(a) Add explanation message*, we follow three steps: (1) match the assert method in the current and parent commits, where the list of arguments in the current commit has one more argument than the parent commit, (2) remove the identical arguments from the list of arguments in the current and parent commits, and (3) check whether the remaining argument is a string. Therefore, our detection rule is able to identify the addition of an explanation message independently of the assert method receiving the string message as the first or last parameter. Differently, the `@Test annotation` and `@Rule annotation` are JUNIT4 constructs. Therefore, the replacements (f) and (g) are dependent on the JUNIT version.

[4]Available at https://github.com/arieslab/TestRefactoringMiner

IV. STUDY I: EVALUATING THE ACCURACY OF TESTREFACTORINGMINER

The *goal* of the first study is to analyze the accuracy of the proposed TESTREFACTORINGMINER, with the *purpose* of understanding the actual capabilities of the tool when employed for the detection of test-specific refactoring actions on test cases written with JUNIT. The main *perspective* is that of software engineering researchers, who are interested in understanding how our approach can support mining software repository studies. More specifically, the goal of the study was mapped onto the following research question (**RQ**):

TABLE I: An overview of the studied systems

Project	Contributors	Stars	Tags	# Commits [2019, 2021]	# Refactored classes
Accumulo	149	971	41	1,628	18,453
Bookkeeper	172	1.7k	57	1008	4,692
Camel	963	4.8k	217	27,169	115,803
Cassandra	410	7.9k	289	1981	33,045
Cxf	197	811	181	2436	1,156
Flink	1,125	21.1k	221	17,801	65,624
Groovy	355	4.8k	248	4,405	23,708
Hadoop	513	13.5k	371	5,481	35,440
Hive	344	4.8k	84	4,050	26,867
kafka	1,009	24.7k	208	4,939	20,005
karaf	151	622	106	1100	10,591
wicket	96	676	303	864	22,630
Zookeeper	192	11.2k	153	677	3,580

RQ$_1$. *What is the accuracy of* TESTREFACTORINGMINER*, in terms of precision and recall?*

To address **RQ**$_1$, we first required to create a dataset reporting actual test-specific refactoring operations applied over the software evolution of open-source systems. Afterwards, we could proceed with comparing the outcome of TESTREFACTORINGMINER against the curated dataset. To design and report our empirical study, we followed the empirical software engineering guidelines by Wohlin et al. [48] and the *ACM/SIGSOFT Empirical Standards*.[5]

A. Context of the study

The *context* of the empirical study was composed of a set of 13 open-source projects from the APACHE SOFTWARE FOUNDATION. Those projects were selected from GITHUB by Kim et al. [49], following the criteria: (i) top 1,000 Java projects ordered by popularity (i.e., stargazer count), (ii) repositories that are not forks, (iii) projects that are above the 90th percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars), and the number of commits. The community has widely studied those projects in the past as they cover different domains, from big data processing and warehousing solutions to distributed databases and programming languages [44], [49]–[52]. In addition, we analyzed a three-year window because studying the evolution of test code on a commit-by-commit basis is expensive. This time window also provides information on refactorings related to recent versions of the testing frameworks (test cases written with JUNIT4 and their migration to JUNIT5). Table I shows an overview of the studied software projects.

B. Building a dataset of test-specific refactoring operations

We implemented a GIT commit history analyzer using JGIT[6] to mine all the commits related to changes in the test files from 2019 to 2021 of the 13 open-source JAVA projects selected. Given a set of files as input, the analyzer selects the commits related to changes in the test files and discards the other commits from further analysis. A commit changes the test file if the involved files have the extension `.java` and have the `[Tt]est(s*)` prefix or suffix. We obtained 12,363 commits with 41,995 changed test files.

As a test file is a common place to receive new test cases, we analyzed whether its commit message suggests refactorings or the co-evolution between test and production code. We manually analyzed all the commit messages of the 12,363 commits to select the relevant ones. As a result, we identified 3,786 relevant commits containing 14,829 changed test files.

Subsequently, we applied stratified random sampling on the test files in the selected commits with a 95% confidence level and 5% confidence interval. As a result, we selected a statistically significant sample of 375 changed test files.

Then, we manually analyzed the statistically significant sample of 375 test files to classify the test-specific refactorings. We analyzed the GIT `diff` between the current and parent commits, representing the changes in the test files. We analyzed whether the lines removed from the parent commit occurs in the test setup, verification, or teardown (parameters in the assertions, annotation tags, and methods signature). Then, we analyzed whether the lines added in the current commit helped organize the code without changing its logic. We did not consider the changes in test files that either add or remove entire test methods or classes as refactorings. In more detail, two coders manually analyzed the changes in the test files to understand the test code problem and classified the test-specific refactorings. Coder A is a Ph.D. candidate, and Coder B is a postdoctoral researcher; both have experience with software quality and they are not authors of the paper—as such, the coders did not have any knowledge on the approach devised in this paper and could provide us with a reliable dataset to compare the output of our approach with. Coder A and Coder B analyzed the same subset of 50 test files and classified 98 instances containing pairs of smelly and refactored test code. The agreement level between the coders was high; they agreed on 196 instances, and each one missed 2 instances (Cohen's kappa = 0.98). Next, a third researcher joined the discussion to classify the four missed instances. The coders added the four missed instances in the final set, totaling 200 instances. Then, Coder A analyzed the remaining 325 test files, generating a dataset containing 729 instances of test refactorings. From this dataset, we generated a sub-dataset containing 417 instances of the most frequent test-specific refactorings.[7]

C. Evaluation of the accuracy of the tool

We run TESTREFACTORINGMINER on the same set of open-source projects selected in the study. Table II presents the distribution of 2,816 refactorings mined by our tool from 2019 to 2021. As the tool did not detect any test-specific refactorings in the projects BOOKKEEPER, HIVE, and KARAF for this time window, we omitted those projects from the table. Most test-specific refactorings performed in the ACCUMULO (46.5%), CAMEL (45.9%), KAFKA (66.9%), and ZOOKEEPER (98.2%)

[5]Available at https://github.com/acmsigsoft/EmpiricalStandards.

[6]The JGIT framework: https://www.eclipse.org/jgit/.

[7]Available at https://figshare.com/s/506813a38d1e3e709533

TABLE II: Diffusion of the test-specific refactorings performed in the studied projects between 2019 and 2021

Projects	(a)	(b)	(c)	(d)	(e)	(f)	(g)	Total
Accumulo	3	3	8	3	113	59	164	353
Camel	19	112	117	15	116	8	329	716
Cassandra	14	21	10	3	-	0	-	48
Cxf	2	65	78	204	-	-	-	349
Flink	11	6	23	1	-	-	2	43
Groovy	2	0	0	1	-	-	-	3
Hadoop	3	-	1	-	-	-	-	4
Kafka	102	135	19	8	111	7	771	1,153
Wicket	-	75	16		-	-	-	91
Zookeeper	-	-	-	1	-	-	55	56
Total	156	417	272	236	340	74	1321	2,816

(a) Add explanation message, (b) Replace reserved words, (c) Split conditional parameters, (d) Replace the not (!) operator, (e) Replace try/catch with assertThrows, (f) Replace @Rule annotation with assertThrows, (g) Replace @Test annotation with assertThrows

TABLE III: Accuracy of the tool's extension in terms of precision and recall per test-specific refactoring type

ID	test-specific refactoring	TP	FN	FP	Precision	Recall
(a)	Add explanation message	63	18	0	1.00	0.78
(b)	Replace reserved words	9	0	0	1.00	1.00
(c)	Split conditional parameters	7	0	0	1.00	1.00
(d)	Replace the not (!) operator	2	0	0	1.00	1.00
(e)	Replace try/catch with assertThrows	40	5	0	1.00	0.89
(f)	Replace @Rule annot. with assertThrows	15	2	0	1.00	0.88
(g)	Replace @Test annot. with assertThrows	237	19	0	1.00	0.93
	Total	373	44	0	1.00	0.92

projects correspond to the *(g) Replace @Test annotation with assertThrows* refactoring. Differently, most test-specific refactorings in the GROOVY (66.7%) and HADOOP (75%) projects correspond to *(a) Add explanation message* refactorings. While most test-specific refactorings performed in the CASSANDRA (43.8%) and WICKET (82.4%) projects correspond to the *(b) Replace reserved words*, in the FLINK (53.5%) project, they correspond to *(c) Split conditional parameters*, and, in the CXF (58.5%) projects, they correspond to the *(d) Replace the not (!) operator* refactorings.

Subsequently, we matched the test-specific refactorings composing our dataset to those the tool detected. Table III presents the accuracy of the extended tool in terms of precision and recall, answering the **RQ**$_1$. The precision corresponds to ($\frac{TP}{TP+FP}$) metric, and the recall corresponds to ($\frac{TP}{TP+FN}$) metric, where TP is the number of true positive instances, FP is the number of false positives, and FN is the number of false negatives. Overall, the tools' extension presents a precision score of 100% and a recall score ranging from 78% to 100%.

While our approach revealed high accuracy scores, it is worth discussing when it failed to inform researchers on the corner cases to consider when using TESTREFACTORINGMINER. It fails to detect the refactorings *(e) Replace try/catch with assertThrows*, *(f) Replace @Rule annotation with assertThrows*, and *(g) Replace @Test annotation with assertThrows* because it expects the developers to use a lambda expression within the `assertThrows` method. The approach checks whether at least one line in the test method

```
184 @Test(expected = IllegalArgumentException.class)
185 public void testWriteIdempotentWithInvalidEpoch(){
... 
195  builder.close();
196 }
```

Listing 1: Test method of Kafka project (Parent commit).

```
184 @Test
185 public void testWriteIdempotentWithInvalidEpoch(){
... 
195  assertThrows(IllegalArgumentException.class, builder::
       close)
196 }
```

Listing 2: Test method of Kafka project (Current commit).

in the parent commit is equal to a line within the lambda expression in the current commit. Listing 1 shows the `testWriteIdempotentWithInvalidEpoch` test method is handling an exception through `@Test annotation` (line 184), which is thrown by the `builder.close()` (line 195) in the parent commit of KAFKA project. Listing 2 shows the refactoring performed by developers, using an *assertThrows* method without a lambda expression (line 195). The tool was unable to match the `builder.close()` in the parent commit with the `builder::close` in the current commit (lines 195), because they should be identical. In other words, the tool was expecting to find `assertThrows(IllegalArgumentException.class, () → builder.close())` in line 195 of the commit.

Besides, the tool also fails for some instances of *(a) Add explanation message*. It occurs because the tool is not able to identify that the code changes refer to a replacement. As it is a problem carried from the previous version of REFACTORINGMINER tool, we need to further investigate how to improve the matching statements for the test code.

> **Summary**$_{RQ1}$**.** TESTREFACTORINGMINER *performs the test-specific refactorings detection with a mean of 100% and 92% precision and recall scores, respectively.*

D. Threats to validity

In the following, we discuss some limitations along with the actions performed to mitigate their effects.

Missing context. REFACTORINGMINER analyzes only the added, deleted, and changed files from two versions. Although that analysis saves computational resources to detect code refactorings, the tool can report incorrect refactoring when it involves unchanged files. Missing context is a limitation we carried on from tool previous versions for detecting code refactorings but does not apply to the new test-specific refactorings.

Nested and composite test-specific refactorings. Our TESTREFACTORINGMINER tool can detect nested test-specific refactorings within a single commit following the implementation of its previous versions. However, we did not implement rules for composite refactorings for this first version.

For example, test-specific refactorings can occur to *(b) Replace reserved words*, *(c) Split conditional parameters* or *(d) Replace the not (!) operator*. Those three replacements are low-level test-specific refactorings, which developers can combine to refactor more complex inappropriate assertions such as the sequence of replacements: $assertTrue(object.equals(null) \rightarrow assertEquals(object, null) \rightarrow assertNull(object)$.

Unsupported refactoring types. We presented and evaluated the detection rules for seven test-specific refactorings. Some test-specific refactorings are defined in the literature and applied in practice [25], [46], [53] *(a) Add explanation message*, *(e) Replace try/catch with assertThrows*, *(f) Replace @Rule annotation with assertThrows*, and *(g) Replace @Test annotation with assertThrows*. Other refactorings we identified in practice were: *(b) Replace reserved words*, *(c) Split conditional parameters*, and *(d) Replace the not (!) operator*. Therefore, we extended REFACTORINGMINER tool with some refactorings defined in formal and gray literature.

Dataset bias. Two experienced external coders built the dataset we employed to address the accuracy of our approach. The coders did not have any knowledge on the inner working of TESTREFACTORINGMINER and could therefore provide us with a reliable source to assess our approach. Nonetheless, the coders still performed manual analysis: although we reduced bias in the dataset construction by assessing the coders' reliability, we could not claim the dataset is unbiased.

V. STUDY II: EVALUATING THE USEFULNESS AND EASE OF USE OF TESTREFACTORINGMINER

The *goal* of the second study was to analyze the viability of TESTREFACTORINGMINER with respect to its usefulness, ease of use, and self-predicted future use from the point of view of researchers in the context of mining software repositories, with the *purpose* to assess the practicality and usefulness of our tool. The *perspective* was of software engineering researchers, who are interested in assessing the maturity of our tool. In particular, the goal of the study aimed at addressing the following **RQ**:

> **RQ$_2$.** *How is the viability of* TESTREFACTORINGMINER *to perform mining test-specific refactorings?*

As the goal of **RQ**$_2$ was concerned with the perception of the adoption of new technology, we conducted the evaluation using the Technology Acceptance Model (TAM) [54]: this is an information systems theory having the goal of eliciting how the stakeholders of a novel technology (software engineering researchers in our case) come to accept and use the technology (TESTREFACTORINGMINER tool in our case). TAM considers three constructs [55]: (1) perceived usefulness to indicate the degree to which a person believes that using a particular system would enhance his or her job performance, (2) ease of use to indicate the degree to which a person believes that using a particular system would be free of effort, and (3) self-predicted future use to indicate the degree to which a person believes that he or she would use a system in the future.

A. Procedure and instrumentation

Target Audience: To ensure valid results, we only selected software engineering researchers with knowledge on software repository mining and code refactoring. More particularly, we involved 15 researchers from our contact network, which we personally invited to perform the study. Their experience with software repository mining ranges between 1 and 5 years, while their experience with refactoring ranges between 1 and 4 years. More importantly, only two participants engaged with REFACTORINGMINER in the past. Therefore, we could gather insights from researchers who were not used to the instrument and not affected by any learning effect.

Questionnaire: We applied a three-step questionnaire:

(1) Part I: Participants' characterization. Participants filled out consent and a characterization form. The goal was to investigate the respondent profile, with information about name, level of education, and experience;

(2) Part II: Participants' tasks performance and perceptions. Participants described their perceptions regarding the positive and negative points of the tool and suggestions for improvements.

(3) Part III: Perceived usefulness, ease of use, and future use. Participants filled out an evaluation form about their perceptions of the tool. The questions in the evaluation form are based on TAM (Table IV), which answers follow a six-point Likert scale to indicate the degree of likelihood that a statement about the usefulness and ease of use of TESTREFACTORINGMINER tool is true (from "Extremely unlikely" (1) to "Extremely likely" (6)). As recommended in previous work [55], we chose a six-point Likert scale to avoid indecisive answers, e.g., the number 3 in a five-point Likert scale.

Pilot test questionnaire: We conducted a pilot study with two participants to improve the instrumentation of this evaluation. One of the participants is experienced in software repository mining and code refactoring, while the other has no experience in performing those activities. We selected them in an effort of gathering opinions from researchers with different expertise, which would have possibly highlighted complementary issues in the way questions were phrased, other than in the way the approach was released. We asked them to perform the tasks and review the survey to ensure the questions were clear and complete. The responses of the pilot study were not considered in the final assessment of the tool.

Tool training and usage: We sent invitations to our target audience via email. In the first step, participants signed the consent term and filled out the participants' characterization form. In the second step, participants read a manual to learn how to install TESTREFACTORINGMINER tool and understand the tasks composing this evaluation. In the third step, participants performed three tasks, as described in Table V, using TESTREFACTORINGMINER tool and filled out the Participants' tasks performance and perceptions form. In the fourth step, participants filled out the usefulness, ease of use evaluation, and self-predicted future use form.

TABLE IV: Question to evaluate the perceived usefulness, ease of use, and self-predicted future use.

Questions regarding perceived "Usefulness" (U):	
U1	Using the TestRefactoringMiner tool in my job, I would be able to mine test-specific refactorings more quickly.
U2	Using the TestRefactoringMiner tool would improve my performance on mining test-specific refactorings.
U3	Using the TestRefactoringMiner tool for mining test-specific refactorings would increase my productivity.
U4	Using the TestRefactoringMiner tool would enhance my effectiveness on mining test-specific refactorings.
U5	Using the TestRefactoringMiner tool would make it easier to mine test-specific refactorings.
U6	I would find the TestRefactoringMiner tool useful to perform mining test-specific refactorings.
Questions regarding perceived "Ease of Use" (E):	
E1	Learning to operate the TestRefactoringMiner tool would be easy for me.
E2	I would find it easy to get the TestRefactoringMiner tool to do what I want it to do.
E3	My interaction with the TestRefactoringMiner tool would be clear and understandable.
E4	It would be easy to become skillful in using the TestRefactoringMiner tool.
E5	It would be easy to remember how to mine test-specific refactorings using the TestRefactoringMiner tool.
E6	I would find the TestRefactoringMiner tool easy to use.
Self-predicted future use (S):	
S1	Assuming the TestRefactoringMiner tool is available on my job, I predict that I will use it on a regular basis in the future.
S2	I prefer using the TestRefactoringMiner tool for mining test-specific refactorings than not using it.

TABLE V: List of tasks the participants performed.

Task	Description
T1	Execute TESTREFACTORINGMINER tool in the project JUNIT-TEAM/JUNIT4 from the first to the last commit of the main branch to collect all refactorings. Locate the test-specific refactoring "Add assert message" and answer the questions: (a) How many test-specific refactorings did the tool report in total? (b) What is the test class name where the test-specific refactoring was applied? (c) What is the line number where the refactoring was applied?
T2	Execute TESTREFACTORINGMINER tool in the project JUNIT-TEAM/JUNIT4 from the main branch's first to the last commit to collect only test-specific refactorings. (a) How many test-specific refactorings did the tool report? (b) Which are the test-specific refactorings reported by the tool? (c) Which is the test-specific refactoring with the highest number of occurrences?
T3	Execute TESTREFACTORINGMINER tool to identify test-specific refactorings in the versions R4.12 and R4.13 of the JUNIT-TEAM/JUNIT4 project. (a) How many test-specific refactorings did the tool report? (b) Which are the test-specific refactorings reported by the tool? (c) What is the test class name where the test-specific refactorings were applied?

TABLE VI: Factor Analysis.

Variable	Usefulness	Ease of use	Future use
U1	**0.94**	0.06	0.00
U2	**1.00**	-0.08	0.02
U3	**0.57**	0.23	0.15
U4	**0.93**	-0.16	0.02
U5	**0.88**	0.14	0.02
U6	**0.89**	0.15	-0.05
E1	0.03	**0.96**	-0.02
E2	0.16	**0.55**	0.36
E3	0.18	**0.6**	0.19
E4	0.34	**0.74**	-0.04
E5	-0.03	**1.00**	-0.09
E6	-0.08	**0.87**	0.25
S1	0.01	0.04	**0.93**
S2	0.49	0.10	**0.48**

B. TAM reliability and factor validity

We calculated Cronbach alpha values to perform a reliability analysis to ensure the internal validity and consistency of the items used for each variable [56]. A Cronbach's alpha reliability level ranging from 0.61 to 0.70 corresponds to the lower limit of acceptability, from 0.71 to 0.80 indicates that the items are homogeneous and measuring the same constant, and from 0.81 to 1.00 indicates a reliable measure [56], [57]. Our results show Cronbach's alpha reliability level over 0.80 for the three TAM variables; 0.972 for perceived usefulness, 0.948 for ease of use, and 0.858 for self-predicted future use.

We performed a factor analysis to check whether the usefulness (U), ease of use (E), and self-predicted future (S) use items form distinct constructs. In our case, the variables are the questionnaire questions (Ui, Ei, and Si) and three factors (U, E, S). The threshold level for sufficient loading is 0.7 [57]. But even lower values are sometimes considered important for a particular factor [57]. Table VI shows that the 14 questionnaire items load on three different factors. Although some items in usefulness (U3) and ease of use (E2 and E3) have values below 0.7, they still load higher on their respective factors.

C. The usefulness, ease of use and self-predicted use of TestRefactoringMiner

Fig. 3 also presents the results of each question related to usefulness, ease of use, and self-predicted use. The participants found that TESTREFACTORINGMINER tool is useful for mining test-specific refactoring in software repositories (mean of 5.07). However, the tool requires some effort to use (mean of 4.5), especially regarding the "Learning to operate TESTREFACTORINGMINER tool would be easy for me".

So far, our data analysis shows that the participants consider TESTREFACTORINGMINER tool useful and easy to use. For example, P11 stated that "the tool facilitates mining tasks by providing a list of analyzed commits and showing the location where the refactoring occurred, if any. The output file is a `.json`, which would make it easier to use other tools to analyze the data and generate charts, e.g. software R". Besides, P15 pointed out that the "tool can be useful to focus only on test-specific refactorings; using the classic version of REFACTORINGMINER would require additional effort to filter out the refactorings performed on production classes".

In order to investigate how their opinions impact user acceptance, we correlated the summative results referring to usefulness, ease of use, and self-predicted future usage [57].

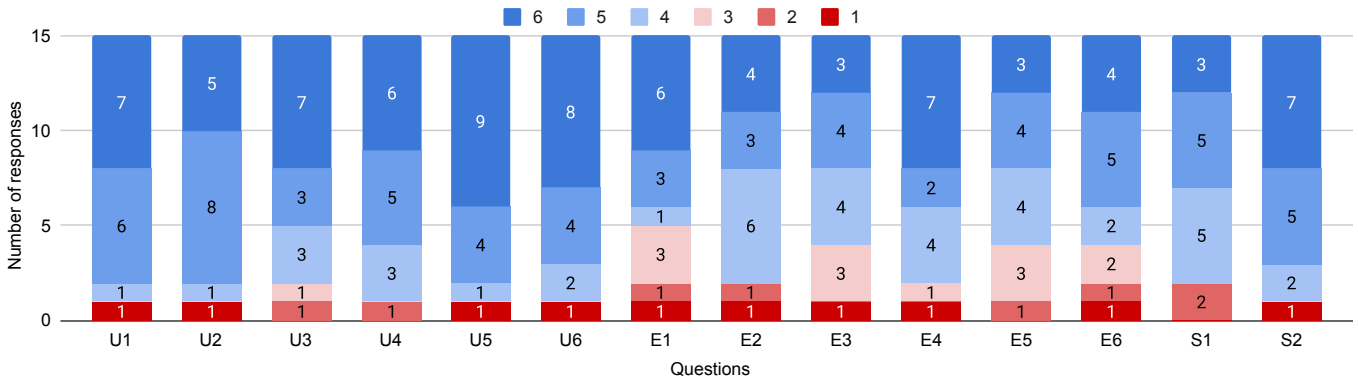

Fig. 3: Usefulness, ease of use, and self-predicted use of TESTREFACTORINGMINER tool. The data is presented on a six-point Likert-scale from (1) Extremely unlikely to (6) Extremely likely.

Ease of use and usefulness are positively correlated (0.43), and both of them are positively correlated with self-predicted future use. The correlation between ease of use and self-predicted future is higher (0.53) than the correlation between usefulness and self-predicted future usage (0.40). It means that the participants' decision-making of using TESTREFACTORINGMINER relies more on how easy or hard it is to use the tool and then its functionalities.

> **Summary**$_{RQ2}$. *The participants consider* TESTREFACTORINGMINER *tool useful to detect test-specific refactorings in software repositories, and easy to use as its results can serve as input to other complementary tools to perform the data analysis. Still, the participants would prefer using a Graphical Interface with filter mechanisms to execute the tool than running it via command line.*

D. Threats to validity

We found two main limitations that might have threatened the results reported in our empirical study.

Tool familiarity. Most participants used REFACTORINGMINER and TESTREFACTORINGMINER tools for the first time during the execution of the empirical study. They only counted on a tutorial to learn how to configure the environment and three examples on how to use the tool. Despite their low experience with the tool, they performed 99.8% of tasks correctly. This further indicates the simplicity of our approach in terms of installation and use.

Process validity. Another threat may be the representativeness of the tasks for mining software repositories. We asked the participants to mine test-specific refactorings with TESTREFACTORINGMINER tool. Then, we asked them to open a `.json` file to answer some questions, aiming to validate the tasks' correctness. On the one hand, participants reported that searching for specific refactorings in a large `.json` file is exhaustive. On the other hand, participants also recognized that a `.json` file could be easily imported to an environment for statistical computing to work with large amounts of data.

VI. CONCLUSION

This paper introduced TESTREFACTORINGMINER, an approach that extends the well-known REFACTORINGMINER to mine seven types of test code refactoring operations performed by developers. The empirical assessment of the approach revealed a precision and recall close to 100%, showing that our approach represents a reliable tool to conduct mining software repository studies. In addition, we surveyed researchers to evaluate the perceived usefulness, ease of use, and self-predicted future use of the tool. We found that TESTREFACTORINGMINER is generally perceived as an actually useful tool, with a few limitations to address and with a high potential to enable further knowledge on test-specific refactoring.

As such, our paper offers to the research community a validated instrument to perform mining software repository studies involving test code: it is our hope that TESTREFACTORINGMINER may actually lead to a number of valuable findings with actionable implications. As for our future work, we aim to continue extending TESTREFACTORINGMINER to mine other test-specific refactorings. In addition, we aim at collecting a larger-scale dataset of test-specific refactorings to assess the recommendations provided by the tool and perform a comparative analysis with alternative approaches able to detect change patterns in the test code.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, Fundação de Amparo a Pesquisa do Estado da Bahia (FAPESB) grants BOL0188/2020 and PIE0002/2022, Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) grant 312195/2021-4, Fundação de Amparo a Pesquisa do Estado de Alagoas (FAPEAL) grants 60030.0000000462/2020 and 60030.0000000161/2022. Fabio is supported by the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090 (TED).

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[3] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[4] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[5] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?" in *WRT*, 2007, pp. 60–61.

[6] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 176–185.

[7] E. Tempero, T. Gorschek, and L. Angelis, "Barriers to refactoring," *Communications of the ACM*, vol. 60, no. 10, pp. 54–61, 2017.

[8] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[9] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, 2020.

[10] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[11] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "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*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 465–475.

[12] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?" in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 186–190.

[13] G. Sellitto, E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, F. Palomba, and F. Ferrucci, "Toward understanding the impact of refactoring on program comprehension," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 731–742.

[14] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, and F. Lima, "Does the introduction of lambda expressions improve the comprehension of java programs?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, ser. SBES '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 187–196.

[15] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.

[16] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[17] E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, and F. Palomba, "Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security," *Empirical Software Engineering Journal*, p. to appear, 2023.

[18] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–120.

[19] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig, "30 years of software refactoring research: a systematic literature review," *arXiv preprint arXiv:2007.02194*, 2020.

[20] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Information and Software Technology*, vol. 140, p. 106675, 2021.

[21] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*. Springer, 2006, pp. 404–428.

[22] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[23] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

[24] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[25] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Centre for Mathematics and Computer Science, NLD, Tech. Rep., 2001.

[26] E. M. Guerra and C. T. Fernandes, "Refactoring test code safely," in *International Conference on Software Engineering Advances (ICSEA 2007)*. New York, NY, USA: IEEE, 2007, pp. 44–44.

[27] G. Meszaros, *xUnit test patterns: Refactoring test code*, ser. Addison-Wesley Signature Series. Upper Saddle River, NJ: Addison-Wesley, 2007.

[28] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 483–494.

[29] J. W. Creswell, "Mixed-method research: Introduction and application," in *Handbook of educational policy*. Elsevier, 1999, pp. 455–472.

[30] F. D. Davis, R. P. Bagozzi, and P. R. Warshaw, "User acceptance of computer technology: A comparison of two theoretical models," *Management science*, vol. 35, no. 8, pp. 982–1003, 1989.

[31] "Evaluating the testrefactoringminer tool," Apr 2023. [Online]. Available: https://figshare.com/s/506813a38d1e3e709533

[32] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 166–177.

[33] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 231–240.

[34] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[35] Z. Xing and E. Stroulia, "The jdevan tool suite in support of object-oriented evolutionary development," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 951–952.

[36] ——, "Refactoring detection based on umldiff change-facts queries," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 263–274.

[37] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: A refactoring reconstruction tool based on logic query templates," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 371–372.

[38] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, p. 269–279.

[39] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin, "Refactorinsight: Enhancing ide representation of changes in git with refactorings information," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1276–1280.

[40] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, "Pyref: refactoring detection in python projects," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 136–141.

[41] M. Dilhara, "Discovering repetitive code changes in ml systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1683–1685.

[42] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in python ml systems," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: ACM, 2022.

[43] M. K. Shiblu, "Jsdiffer: Refactoring detection in javascript," Master's thesis, Concordia University Montréal, Québec, Canada, 2022.

[44] D. J. Kim, N. Tsantalis, T.-H. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 62–73.

[45] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 193–202.

[46] E. Soares, M. Ribeiro, G. Amaral, R. Gheyi, L. Fernandes, A. Garcia, B. Fonseca, and A. Santos, "Refactoring test smells: A perspective from open-source developers," in *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, 2020, pp. 50–59.

[47] V. Veloso, "Fork of refactoringminer," 2021, Accessed on 05.01.2023. [Online]. Available: https://github.com/victorgveloso/RefactoringMiner

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[49] D. J. Kim, T.-H. P. Chen, and J. Yang, "The secret life of test smells-an empirical study on test smell evolution and maintenance," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–47, 2021.

[50] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 2012–2031, 2019.

[51] R. Vieira, A. da Silva, L. Rocha, and J. a. P. Gomes, "From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 80–89.

[52] D. J. Kim, "An empirical study on the evolution of test smell," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 149–151.

[53] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. M. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date," *IEEE Transactions on Software Engineering*, 2022.

[54] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS quarterly*, pp. 319–340, 1989.

[55] M. A. Babar, D. Winkler, and S. Biffl, "Evaluating the usefulness and ease of use of a groupware tool for the software architecture evaluation process," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 430–439.

[56] E. G. Carmines and R. A. Zeller, *Reliability and validity assessment*. Sage publications, 1979.

[57] O. Laitenberger and H. M. Dreyer, "Evaluating the usefulness and the ease of use of a web-based inspection data collection tool," in *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*. IEEE, 1998, pp. 122–132.