Does Refactoring Break Tests and to What Extent?

Yutaro Kashiwa*, Kazuki Shimizu*, Bin Lin†
Gabriele Bavota†, Michele Lanza†, Yasutaka Kamei*, Naoyasu Ubayashi*,

*Kyushu University, Japan — †Software Institute – USI, Lugano, Switzerland

***Abstract*—Refactoring as a process is aimed at improving the quality of a software system while preserving its external behavior. In practice, refactoring comes in the form of many specific and diverse refactoring *operations*, which have different scopes and thus a different potential impact on both the production and the test code. We present a large-scale quantitative study complemented by a qualitative analysis involving 615,196 test cases to understand how and to what extent different refactoring operations impact a system's test suites. Our findings show that while the vast majority of refactoring operations do not or very seldom induce test breaks, some specific refactoring types (e.g., "RENAME ATTRIBUTE" and "RENAME CLASS") have a higher chance of breaking test suites. Meanwhile, "ADD PARAMETER" and "CHANGE RETURN TYPE" refactoring operations often require additional lines of changes to fix the test suite they break. While some modern IDEs provide features to automatically apply these two types of refactoring operations, they are not always able to avoid test breaks, thus demanding extra human efforts.**

I. INTRODUCTION

Refactoring is a good practice, leading to improved code quality, higher developer productivity, and less error prone code [1] [2]. This intuition is supported by previous studies [3][4]. By definition, refactoring does not change the external behavior of a system. However, refactoring can break test suites because it can modify the internal behavior which is tested in the test suites.

Test suites can reach a considerable size in modern software development. Fixing them can be an extensive and time-consuming task [5]. Vassallo et al. [6] surveyed continuous refactoring barriers and indicated that one of the reasons developers hesitate to refactor code is the possibility to break test suites. The problem is exacerbated by the recent adoption of continuous integration and deployment practices and in general fierce market competition [7].

Many studies [8][9][10][11] investigated the impact of refactoring on test code. Rachatasumrit et al. [8] claimed that half of the broken test code involves refactoring edits. Still, it is not clear how refactoring breaks test suites because they did not conduct a qualitative analysis. Indeed, in their work, they indicate that the refactoring operations involved in the failed tests might not be the cause. To unveil the impact of refactoring on test suites, this study sets out to address the following central question:

"Does refactoring break tests and to what extent?"

We conducted a conceptual replication[12][13] of the study by Rachatasumrit and Kim [8], exploiting state-of-the-art mining techniques to run a more fine-grained analysis. Specifically, we use commit histories rather than released versions. Such a change in granularity reduces the size of each change we need to inspect, which makes it easier to identify the cause of the failures during qualitative analysis. However, applying dynamic trace tools as in the mentioned study is expensive since it requires long execution time, heavy memory usage, etc. [14]. Running such an analysis to every revision in each studied repository further worsens the issue. Leveraging modern technology, we exploit Kubernetes [15] to run our analyses through several computational instances in parallel.

Our experiments involve 615,196 test methods taken from 8 projects, and a total of 22,000 computation hours in the containers. The research questions and results are as follows:

RQ_1: What types of refactoring break test suites?

Most of the refactoring types, as expected, are not observed to break test methods but *Compiler-Errors* can be triggered by refactoring operations that modify method signatures (e.g., MOVE CLASS, ADD PARAMETER). *Failures* are rarely caused by refactoring, but sometimes they do occur when CHANGE RETURN TYPE and ADD PARAMETER are performed. A previous study [8] reported that many test methods are broken by refactoring operations. We found that only 2.5% of refactoring operations break test suites.

RQ_2: What is the magnitude of the fix triggered by different types of refactoring on test methods?

We observed that the efforts needed to fix test breaks triggered by refactoring vary for different refactoring types. However, while ADD PARAMETER and CHANGE RETURN TYPE seldom break tests, they are more likely to have a wider impact than other types of refactoring, requiring 4-12 lines of changes to fix the tests they break.

Discussion: Can IDEs help in reducing the chances of breaking tests during refactoring activities?

IDEs automate the mechanical part of refactoring by, for example, consistently changing all references to a variable when its identifier is renamed. We found that they can help to avoid test breaks triggered by refactoring involving rename or move actions. However, they did not prevent test breaks triggered by CHANGE RETURN TYPE, CHANGE PARAMETER TYPE, or ADD PARAMETER.

Paper Organization. Section II introduces our research questions and discuss the related literature. Section III describes our study design, while Section IV discusses the achieved results, thus answering our research questions. In Section V we conduct additional analyses and discuss the broader implications of our results. Finally, Section VI summarizes our findings and discusses future work.

II. Background and Related Work

When developers modify their source code, they run their test suites to detect regressions. Past studies [16][17] have developed approaches to detect the test methods affected by changes in the source code since exercising the whole test suite can be time-consuming [18][19][20]. These approaches can precisely identify test methods that should be run, which reduces the time for running test suites. However, the modified source code is not always in a state so that it can be built. It has been reported that changes in the production code break test suites in many cases [8][21].

Tang et al. [21] examined what percentage of bug-fixing changes cause test failures. They reported that 48.7% of bug-fixing changes would break regression test suites. Thus, developers will recognize that they need time to fix test suites after modifying the production code, which disrupts their development schedule.

Even refactoring, which should not change the behavior of a system, is known to break test suites. Rachatasumrit and Kim [8] examined what proportion of failed tests are relevant to refactoring edits. They showed that half of the failed tests involve refactoring edits in the production code. The previous studies deepen the understanding of the impact that changes have on test suites, which fosters the development of automatic repair tools for test suites. However, there are still missing pieces of evidence that we aim at building in our study:

1) It is uncertain how refactoring breaks test suites: Rachatasumrit and Kim [8] found that refactoring operations might lead to failed tests. It is however unclear whether the refactoring edits are the real causes of the test failures, since they are often part of tangled commits also involving other types of changes. To identify the real cause, a manual inspection would be needed. However, such an inspection would be challenging on those data since the refactoring edits have been extracted between releases, and the numbers of both refactoring and non-refactoring related code changes involving a test method are tremendous.

To overcome this issue, we mine the complete change history of the studied systems at commit level, thus obtaining much smaller changes. Also, we study the impact of a single and isolated refactoring operation on the test methods exercising the refactored production code. This has been possible thanks to novel mining tools developed in the last years and allows our study to be more fine-grained and involving a higher number of data points. Also, the precise data extracted (i.e., a single refactoring operation impacting tests) allow for a simpler manual inspection that can validate the impact of refactoring on the test.

This can be summarized by the following research question:

RQ_1: What types of refactoring break test suites?

2) The fixing cost of a broken test is not a constant: The tests broken by different refactoring types may trigger a different fixing cost.

For example, Adding a Parameter to a method may require the writing of several new lines in the test method to prepare the additional input represented by the new argument. Instead, a Rename Class refactoring may only impact a single line in the test method (i.e., the change to the class name).

A study by Elish and Alshayeb [10] estimated the impact of refactoring operations on the testing effort required by the refactored code components. Such an impact is estimated in terms of internal quality metrics that are measured before and after the refactoring and is based on the findings by Bruntink and van Deursen [22] showing that changes in internal quality metrics result in changes in the testing effort. Elish and Alshayeb found that Encapsulate Field and Extract Method increase the testing effort. Differently from our study, they do not assess the size of the fix triggered in test code by refactoring operations.

Besides the work by Elish and Alshayeb [10], it is worth mentioning the several previous studies analyzing the impact of refactoring on quality attributes. Stroggylos and Spinellis [23] mined refactoring operations from version control system logs of three open-source libraries with the goal of studying their impact on the values of nine object-oriented quality metrics. Their results show the possible negative effects that refactoring can have on some quality metrics. Similarly, Chávez et al. [24] and Cedrim et al. [25] reported that refactoring not always result in an increase of code quality. Such a research question has also been investigated by Szoke et al. [26], who reported that small refactoring operations performed in isolation rarely impact software quality, while a high number of refactorings performed in a block can result in notable code quality improvement.

Moser et al. [27] conducted a case study in a close-to industrial environment to investigate the impact of refactoring on the productivity of an agile team. The achieved results show that in the context of mobile apps development, refactoring increases developers' productivity.

Previous work also analyzed the extent to which refactoring activities induce faults [28]. Authors showed that refactorings involving hierarchies (e.g., *Push Down Method*) induce faults more frequently than others that are likely to be harmless in practice. This finding highlights the strong bond between refactoring and (regression) testing, that also pushed researchers to propose techniques aimed at recommending refactoring solutions (e.g., to remove antipatterns) by keeping low the testing effort required after refactoring [29]. On a related research thread, it is also worth mentioning the study by Sabané et al. [30], that showed the higher testing cost of classes affected by antipatterns.

Related to the above-described work, we formulate the following research question for our study:

RQ_2: What is the magnitude of the fix triggered by different types of refactoring on test methods?

TABLE I
SUMMARY OF THE SYSTEMS UNDER STUDY.

Project	Domain	History			Latest snapshot		
		Commits	Refactoring commits	Refactoring instances	LOC (Production)	LOC (Test)	Test methods
COMMONS-IO	IO functional library	2,740	444	4,064	13,736	24,714	1,310
SPRING	SQL mapping framework	1,615	155	782	1,924	3,933	120
JODA-BEANS	Code generator framework	840	279	3,515	17,068	34,185	444
JSOUP	HTML library	1,386	289	1,358	12,570	9,201	725
SPARK	Web framework	1,062	309	1,789	6,253	5,045	320
LITTLEPROXY	HTTP proxy	1,003	252	1,616	4,180	4,665	55
RXJAVA-JDBC	Database client library	850	186	849	4,611	3,330	74
SPOON	Program analysis library	3,278	878	9,564	64,038	44,960	1,938

III. STUDY DESIGN

The goal of our study is to investigate the extent to which refactoring operations break the related test suites, by answering the two research questions we formulated in Section II. The basic idea behind our study is to simulate a regression testing scenario during a refactoring process: Once the code is refactored, developers can check whether bugs have been introduced by running (part of) the test suite. To reproduce such a scenario, we take the production code refactored in commit c_i and we run on it the test suite before any change triggered by the refactoring has been implemented by the developer. To do so, we run on the production code in c_i the test suite in the previous revision (c_{i-1}). In particular, as what was done in the previous studies [8][21], we first checkout the two snapshots c_i and c_{i-1}. Then, we replace the test directory in c_i with the one in c_{i-1}. Finally, we identify tests broken (i.e., tests that fail or that exhibit compilation errors) by the refactoring operation.

Below we detail our study design, explaining how we collect and analyze the data to answer our research questions.

A. Context Selection

We selected as context for our study eight projects from GitHub (summarized in Table I) satisfying the following criteria.

- *(Criterion 1) Java system:* In this study, we use a state-of-art refactoring mining tool [31] that only works on Java code.
- *(Criterion 2) JUnit:* There are many testing frameworks for Java systems, such as JUnit [32] and TestNG [33]. We target projects that use JUnit 4 or 5, which is an open-source testing framework and a *de facto* standard for Java projects [34].
- *(Criterion 3) Maven project:* We must compile production and test code and run test code for each commit, which is computationally expensive. In this study, we target maven projects to easily automate these processes. Maven [35] is a popular build tool widely adopted for Java systems.
- *(Criterion 4) Simple Project:* Projects sometimes have sub-projects depending on each other. When a sub-project is broken, the project build might fail. To avoid this, we decided to exclude projects that contain sub-projects.

B. Data Collection

To determine which test methods are affected by each refactoring operation performed in a software repository, it is necessary to verify if each test method exercises the production code where refactoring operations have been performed. Such a procedure, depicted in Figure 1, consists of three main steps that we detail in the following: "Refactoring Detection", "Test Run", and "Impact Analysis".

Refactoring Detection. We detect refactoring operations performed in the production code of each commit of the studied repositories. This has been done by using RefactoringMiner (ver. 2.0.3) [31], the state-of-the-art refactoring detector. RefactoringMiner provides, together with each detected refactoring, also exact information about the code lines impacted by it. Note that the location of the refactoring can be a single file (e.g., in the case of rename operations) as well as multiple files (e.g., a method moved from a source to a target file). RefactoringMiner has been shown to be able to detect 55 types of refactoring operations with higher precision and recall than other tools [31][36].

We run RefactoringMiner on every commit in all branches[1] of the cloned repositories to obtain the type and location of the performed refactorings. Note that common commits present in multiple branches are considered only once. Merge commits are excluded since they generate duplicated results.

Test Run. To determine whether a specific test method was broken by refactoring edits, we must know which lines in the production code are exercised by the test method. Indeed, in our study, we consider a refactoring as responsible for breaking a test method t only if there is an overlap between the lines of production code impacted by the refactoring (as reported by RefactoringMiner) and the ones exercised by t. To establish such a link, we employ a dynamic execution trace method that provides us with the production code paths executed by each test method.

To run tests with Maven, the Surefire plugin is needed. Most of the maven projects have it, but some projects do not (they test the product on their local environments without continuous integration). We inserted the plugin setting in the build files (`pom.xml`) to automate testing.

[1]We decided to use all branches because Kovalenko et al. [37] claimed that using only partial history distorts results.

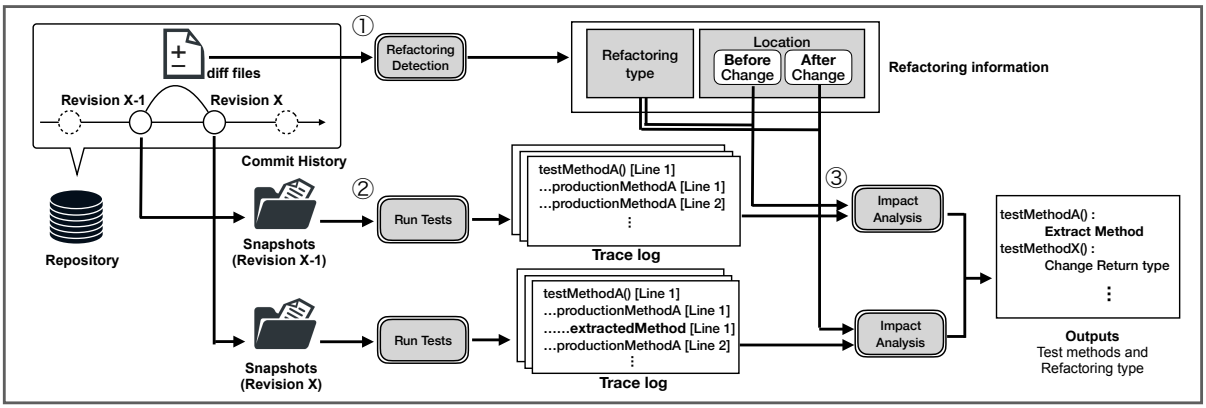

Fig. 1. Overview of the study design. Goal: Identifying if each test method in a repository is affected by a refactoring operation. Our approach extracts two snapshots and runs tests on them, using a dynamic execution analyzer, and examines if there are refactoring edits impacting the traces.

By default, maven testing stops if a test method fails. Since we need to run all tests even if a test method fails, we enabled the "testFailureIgnore" option in the Surefire plugin.

Also, running the tests requires to identify them in the analyzed repositories. If the `pom` file of a project did not contain specified production and test directories, maven default production and test directories (i.e., "src/main/java" and "src/test/java") are used. Test methods are identified using the `@Test` JUnit annotation. We regard these test methods as test cases and do not distinguish unit tests or system tests.

Dynamic trace execution methods are known for being costly. To reduce the required time, memory, and disk space we use SELogger [38]. SELogger collects most of the execution paths but omits redundant paths (i.e., for/while loops). We add the dependency on SELogger into the build file of the studied projects so that SELogger can be executed during test runs. With the updated file, we first compile the test code for each commit on which we detected at least one refactoring. Tests are run both on the snapshot before and after the refactoring (i.e., c_i and c_{i-1}). If a test method causes compilation errors, we remove it and the remaining test methods get compiled again. The removed methods are recorded and this information is used to count compiler errors in RQ_1. We repeat this removing and compiling process until all of the tests are successfully compiled. After that, we execute the tests remaining in test suites. More specifically, we run each test method in isolation rather than the whole test suite to avoid Out of Memory Exceptions during log trace collection.

We skip commits that include errors in production code (e.g., due to dependency problems, compiler errors). On the remaining commits, we executed 615,196 test methods. Then, we excluded test methods failing in c_{i-1}: If a refactoring has been performed in commit c_i and the related tests in the revision c_{i-1} were failing even before the refactoring, those must be excluded, since we want to identify test breaks caused by the refactoring operation, and not already present in the system. This filter removed 3,411 test methods.

Impact Analysis. After running tests, the execution trace for each revision is generated, which provides us with the lines the tests exercise in the production code. In addition, RefactoringMiner provides us with the location of the production code where refactoring is performed. Matching the two outputs, we identify whether refactoring edits impacted any of the paths exercised by any of the test methods. Figure 2 outlines how we match refactoring edits to test case execution paths.

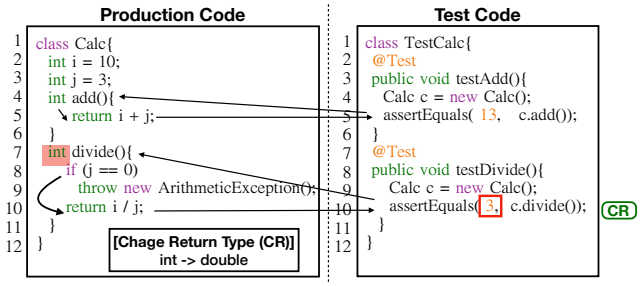

Fig. 2. Specifying refactoring impact: A red shade represents a refactoring. The refactored line is used only by testDivide(). After refactoring, the assertion in testDivide should be modified from the int value of 3 to the double value of 10/3. The arrows represent the order of statements invoked by each test method.

In Figure 2, a CHANGE RETURN TYPE refactoring is performed in *Calc.divide()*. Since the refactored line is exercised by *testDivide()*, the method signature (i.e., package, class, method name, parameter types, and line number) is recorded in the trace log. Thus, *testDivide()*, at line 10, is determined to be affected by the refactoring, whereas *testAdd()* is not because it does not invoke *divide()*. If attributes are impacted by a refactoring, we only identify as impacted test methods that use those attributes in the execution traces to reduce false positives.

We apply this approach to the two snapshots before and after changes because several types of refactoring (e.g., ADD METHOD ANNOTATION) are recorded only in one of the two snapshots by RefactoringMiner as they do not modify any lines in one snapshot. In contrast, the changes caused by other refactoring types (e.g., INLINE VARIABLE) are recorded by RefactoringMiner for both snapshots.

We identify them as the same refactoring by using their hash values given by RefactoringMiner.[2] Thanks to the performed analysis, we know for each commit the lines in the test methods affected by the detected refactoring operations.

Once this information is known, we also excluded from our study test methods affected by multiple refactoring operations, because we cannot identify the exact refactoring operation that possibly resulted in a broken test. We keep the cases in which a single refactoring impacted multiple lines in the production code, since we can still identify, in these cases, the specific refactoring operation causing the test to break. This filter removed 6,314 test methods.

Our data collection process requires considerable computational resources (about 3 hours per commit). For this reason, we employed Kubernetes to parallelize the computation. We deployed Kubernetes on 30 nodes of m5.2xlarge instances (8 vcpu and 32GB memory) of Amazon Web Services and 10 nodes on our local servers, running our pipeline for about one month. The program used for data collection is available on GitHub (https://github.com/posl/DoesRefactoringBreakTest).

C. Data Analysis

To answer our research questions, we replace the test directory in c_i with the one in c_{i-1}. Finally, we identify tests broken (i.e., tests that fail or that exhibit compilation errors) by the refactoring operation using the previously extracted information. Then, two of the authors manually and independently inspected each test method that resulted in test breaks (i.e., compilation error or test failure) to make sure they were actually caused by the refactoring. The two authors classified in the same way 83.6% of the inspected test breaks, with a Cohen's kappa coefficient of 0.66, which demonstrates a substantial agreement [39]. To solve the disagreements, a third author inspected the conflicting cases, discussing them with the first two authors and deciding on the final outcome (i.e., the test was or was not broken by the refactoring operation).

Differently from previous studies [8] [21], we also classify the test breaks into three subcategories:

***Compiler-Error*:** The production code after the refactoring can be built, but the test code (related test methods) cannot compile (e.g., new arguments have been added to the tested method).

***Runtime-Error*:** The test method suddenly terminated during its running (e.g., NullPointerException).

***Failure*:** The test method received unexpected values from the production code after refactoring.

For each category of test break, we report the number of test breaks that are involved in each type of refactoring. In addition, the rates (i.e., compiler-error rate, runtime-error rate, and failure rate) are calculated to identify the refactoring types that are more likely to involve test breaks (if any).

[2]RefactoringMiner's Java API provides a hash value for each refactoring instance to identify each of them, based on the refactoring type name, locations before and after refactoring, etc.)

For example, the failure rate for test methods involving refactoring α is calculated as follows:

$$FailureRate_{\alpha} = \frac{\textit{failures involving refactoring } \alpha}{\textit{all test methods affected by refactoring } \alpha}$$

While the above data analysis is sufficient to answer RQ_1, to address RQ_2 we compute the median of changed lines in broken test methods. For each refactoring type, we calculate the following metric.

***Changed lines*:** We extract file diffs with the "`git diff`" and sum the numbers of added and deleted lines in each test method impacted by refactoring and other relevant methods in test code (e.g., util methods). We used the histogram algorithm to generate diffs because Nugroho et al. [40] showed that the algorithm can precisely assess the impacted lines.

IV. Results

Below we detail our following findings with examples:

- No test suites involving refactoring of local variables failed.
- Refactoring operations changing method signatures and return types are the most likely to trigger *Compiler-Errors*.
- No *Runtime-Error* caused by refactoring operations could be identified.
- It is rather rare for refactoring operations to directly introduce *Failures*.
- The number of lines needed to fix test breaks triggered by refactoring varies for different refactoring types, with CHANGE RETURN TYPE being the most likely to require demanding fixes.

A. RQ_1: What types of refactoring break test suites?

Table II shows the number of test methods involved in our study for each type of refactoring and the frequency with which each refactoring operation type results in passed or broken tests. The refactoring types are grouped into three categories – "Local Variable", "Method", and "Class" – based on the entity on which a refactoring operation is performed. Since, as any automated procedure, our data extraction pipeline can result in false positives, all data in our study has been manually validated. In particular, the manual inspection aimed at removing cases in which the breaking test change was not the refactoring operation. The manual validation was performed by three authors: Each instance was independently inspected by two authors and a third one was in charge of solving conflicts (16.4% of cases). The manual validation resulted in the exclusion of seven false positives, one related to *Compiler-Errors* and six to *Failures*. These false positives are already excluded from Table II. In addition to that, ten test methods encountered *Runtime-Errors* for which we could not identify the root cause of the error. We also removed them from our analysis, leading to no remaining *Compiler-Error*. We also exclude the errors that are caused by other factors (e.g., errors due to runtime-environments, invalid dependencies, flakiness [41]).

TABLE II
RQ1: THE NUMBER OF TEST METHODS THAT ARE INVOLVED IN REFACTORING AND RESULT IN EACH TYPE OF TEST BREAKS. THE REFACTORING OPERATIONS THAT APPEAR LESS THAN 10 TIMES ARE NOT SHOWN DUE TO SPACE LIMITATIONS. THE LAST ROW INCLUDES THOSE INSTANCES OMITTED IN THE TABLE, THUS DOES NOT EQUAL THE SUM OF PREVIOUS ROWS.

Category	Refactoring Type	#methods	#PASS	#COMPILER ERROR	#FAILURE
Local Variable	EXTRACT VARIABLE	1,068	1,068	0 (0.0%)	0 (0.0%)
	REPLACE VARIABLE WITH ATTRIBUTION	608	608	0 (0.0%)	0 (0.0%)
	RENAME VARIABLE	458	458	0 (0.0%)	0 (0.0%)
	CHANGE VARIABLE TYPE	145	145	0 (0.0%)	0 (0.0%)
Method	CHANGE RETURN TYPE	498	490	**7 (1.4%)**	**1 (0.2%)**
	EXTRACT METHOD	318	318	0 (0.0%)	0 (0.0%)
	EXTRACT AND MOVE METHOD	312	312	0 (0.0%)	0 (0.0%)
	CHANGE PARAMETER TYPE	301	300	**1 (0.3%)**	0 (0.0%)
	RENAME METHOD	276	251	**25 (9.1%)**	0 (0.0%)
	ADD PARAMETER	195	180	**13 (6.7%)**	**2 (1.0%)**
	RENAME PARAMETER	114	114	0 (0.0%)	0 (0.0%)
	ADD METHOD ANNOTATION	75	75	0 (0.0%)	0 (0.0%)
Class	MOVE CLASS	567	526	**41 (7.2%)**	0 (0.0%)
	ADD CLASS ANNOTATION	171	171	0 (0.0%)	0 (0.0%)
	RENAME ATTRIBUTE	125	92	**33 (26.4%)**	0 (0.0%)
	EXTRACT ATTRIBUTE	99	99	0 (0.0%)	0 (0.0%)
	CHANGE ATTRIBUTE TYPE	57	57	0 (0.0%)	0 (0.0%)
	RENAME CLASS	32	22	**10 (31.2%)**	0 (0.0%)
	All	5,478	5,343	132 (2.4%)	3 (0.1%)

Finding 1. No test suites involving refactoring of local variables failed. From Table II we can see that while over 2,000 test methods involving refactoring were performed on local variables, none of them led to any error or failure. This finding is partially in contrast with what has been reported in previous work [8], in which refactoring operations involving local variables were reported as likely to introduce bugs. Clearly, a bug could still be introduced without breaking changes in the test suite (e.g., in cases in which the test suite, despite exercising the lines of code involving the variable, is particularly weak in terms of asserted behavior). However, a possible explanation for such a different finding is also the lack of manual inspection and the use of changes between releases in the previous work, which may classify a refactoring as responsible for introducing a bug despite that other changes tangled to the refactoring might be the bug-triggering event.

Finding 2. Refactoring operations changing method signatures and return types are the most likely to trigger *Compiler-Errors*. Seven types of refactoring operations, including CHANGE RETURN TYPE, CHANGE PARAMETER TYPE, RENAME METHOD, ADD PARAMETER, MOVE CLASS, RENAME ATTRIBUTE, and RENAME CLASS, resulted in *Compiler-Errors*. Most of them, except CHANGE RETURN TYPE and RENAME ATTRIBUTE, change method signatures (i.e., package name, file name, class name, method name, and/or parameter types) [42]. Changing the method signature is indeed an obvious breaking change when it comes to locations of the code (such as tests) invoking the refactored method. Moreover, changes of return types can also lead to type inconsistency, while not impacting method signatures. An example of test broken by CHANGE RETURN TYPE can be seen in Snippet 1: A developer changed the return type of the method "`removeSdchEncoding`" from "`HttpHeaders`" to "`void`", while no other changes were made.

As the original test method still expected to receive an instance of "`HttpHeaders`", a *Compiler-Error* occurred. Therefore, the developer had to perform "`removeSdchEncoding`" outside of the assertion statement and use the "`headers`" in the assertion instead (Snippet 2).

Snippet 1. "Change Return Type" refactoring causing a compiler error

```
- public static HttpHeaders removeSdchEncoding(
-                            HttpHeaders headers) {
+ public static void removeSdchEncoding(
+  HttpHeaders headers) {
```

Snippet 2. Fix for compile errors due to "Change Return Type" in test code

```
- ...
- assertEquals(expectedEncodings,
-           ProxyUtils.removeSdchEncoding(headers)
-           .get(HttpHeaders.Names.ACCEPT_ENCODING));
+ ...
+ ProxyUtils.removeSdchEncoding(headers);
+ assertEquals(expectedEncodings,
+    headers.getAll(HttpHeaders.Names.ACCEPT_ENCODING));
  ...
```

Finding 3. No *Runtime-Error* caused by refactoring operations could be identified. Throughout our study, only ten *Runtime-Errors* occurred and we could not identify the cause of the error. Other than that, we did not find any *Runtime-Errors* related to refactoring. The reason might be that *Runtime-Errors* were resolved by developers when modifying production code. After developers modified the production code, they might ran the test suites. If any *Runtime-Error* was spotted during the exercise of test methods, they might tend to further modify the production code until no *Runtime-Error* occurred before committing the changes. However, such cases could only be identified through a more fine-grained analysis capturing code editing in the IDE. The commit-level granularity of our study does not allow to unveil them.

Finding 4. It is rather rare for refactoring operations to directly introduce *Failures*. *Failures* were observed in only three test methods, with all of the corresponding to refactoring operations at the method level (i.e., CHANGE RETURN TYPE and ADD PARAMETER). An example of test failure caused by CHANGE RETURN TYPE can be found in Snippet 3.

Snippet 3. Example of changes causing failures

```
-  public List<Element> previousElementSiblings() {
+  public Elements previousElementSiblings() {
```

Snippet 4. Example of test methods detecting failures

```
@Test
 public void testPreviousElementSiblings() {
   Document doc = Jsoup.parse(
      "<ul id='ul'>" +
      "<li id='a'>a</li>" +
      "<li id='b'>b</li>" +
   ...
- List<Element> elementSiblings =
-                    element.previousElementSiblings();
+ Elements elementSiblings =
+                    element.previousElementSiblings();
   ...
  assertEquals("a", elementSiblings2.get(0).id());
  assertEquals("b", elementSiblings2.get(1).id());
```

A developer changed the return type of the method "`previousElementSiblings()`" from "`List<Element>`" to "`Elements`". As "`Elements`" extends "`List<Element>`" and it was already implemented before the refactoring, no *Compiler-Error* was raised. However, for an instance of the type "`Elements`", the method "`previousElementSiblings()`" will produce a list where siblings are ordered by its distance from the current node, while "`List<Element>`" will generate a list of siblings keeping their original order as child nodes of their common parent. Therefore, the different behavior of the method "`previousElementSiblings()`" after refactoring resulted in the *Failure*. Interestingly, developers only fixed this issue in a later commit after the refactoring got merged into the codebase.

RQ_1: While a previous study [8] showed that many refactoring-related commits break tests, our manual inspection indicates that refactoring operations only account for 2.5% of the real cause of test breaks. Refactoring operations changing method signatures are more likely to cause *Compiler-Errors* in test code, and most of the time refactoring does not directly trigger *Failures*.

B. RQ_2: What is the magnitude of the fix triggered by different types of refactoring on test methods?

Figure 3 depicts, for each refactoring type for which we observed broken tests, the distribution of the numbers of changed lines in each test method as a consequence of refactoring operations in relevant production code. In these violin plots, the thickness of the outer layer represents the probability density of the plotted values. In the center of each violin plot, the white dot represents the median, while the thick black bar represents the interquartile range.

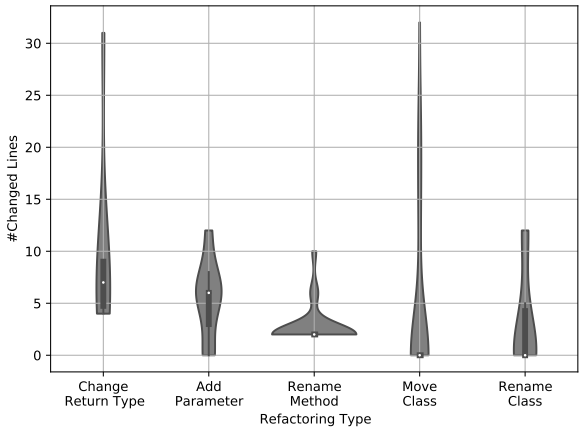

Fig. 3. Distribution of changed lines in test suites to fix test breaks introduced by refactoring operations. We do not show CHANGE PARAMETER TYPE here as it only occurred once in our study.

Finding 5. The number of lines needed to fix test breaks triggered by refactoring varies for different refactoring types, with CHANGE RETURN TYPE being the most likely to require demanding fixes. As it can be seen from Figure 3, most of the issues caused by refactoring can be fixed within 15 changed lines in a test method. The median number of changed lines needed for CHANGE RETURN TYPE is 7, which is the highest among all types of refactoring. CHANGE RETURN TYPE often requires developers to modify the lines around the refactored methods, which is illustrated in Snippet 6. In this case, similarly to the one previously discussed, developers changed the return type of the method "`nextElementSiblings`" from "`List<Element>`" to "`Elements`" (Snippet 5), thus a different approach is needed to verify if the element has any next sibling.

Snippet 5. Example of Change Return Type refactoring

```
-  public List<Element> previousElementSiblings() {
+  public Elements previousElementSiblings() {
```

Snippet 6. Example of fixes required by Change Return Type, modifying the approach to assert values

```
 List<Element> elementSiblings4 = div.
   nextElementSiblings();
- try {
-   Element elementSibling = elementSiblings4.get(0);
-   fail("This element should has no next siblings");
- } catch (IndexOutOfBoundsException e) {
- }
+ assertEquals(0, elementSiblings4.size());
```

Most test breaks caused by ADD PARAMETER can be fixed by modifying around five lines of code in test methods. When manually inspecting how these broken tests were resolved, we found that 73.3% (11/15) of them were fixed by adding null values to the parameter list, while for the remaining 26.7% of cases, developers had to manually select the correct argument to pass. The majority of the test breaks caused by RENAME METHOD requires less than five changed lines for the fix.

As test methods usually invoke the renamed methods several times, during our manual inspection, we noticed in almost all cases (∼96%) the lines changes in the test methods just required to rename the method to invoke.

MOVE CLASS and RENAME CLASS are least likely to require changes in test code: the medians of changed lines are 0 for these two types of refactoring. However, in some rare cases, significant changes are needed for MOVE CLASS. To better understand how the impact of different refactoring types differs from each other, we applied the Mann-Whitney U-test with a Bonferroni correction to all the pairs of distributions of changed lines. We also measured effect size r from the z-score of the U-test. As a result, statistically significant differences (p-value $< 0.01/10$) can be found between the pairs of CHANGE RETURN TYPE and RENAME METHOD with a large effect size (r=0.63), as well as between RENAME METHOD and MOVE CLASS with a medium effect size (r=0.43). The result is in line with our findings from Figure 3.

RQ_2: ADD PARAMETER and CHANGE RETURN TYPE seldom break tests but can have a substantial impact on the test code, requiring 4-12 lines of changes to fix the test suite.

V. DISCUSSION

In the following we detail, complemented by examples, the following findings:

- IDEs do not help to avoid test breaks when the methods in production code are subject to ADD PARAMETER refactoring, since an additional input is required by the refactored method, triggering changes in the test code that must be manually implemented.
- When CHANGE RETURN TYPE or CHANGE PARAMETER TYPE is performed, the IDEs are unable to examine whether the impacted instance still has valid callees or has a compatible data type as a parameter, thus not helping to avoid test breaks.

A. Discussion: Can IDEs help in reducing the chances of breaking tests during refactoring activities?

Modern IDEs provide great support for refactoring source code. While they are not meant to recommend refactoring operations, IDEs such as IntelliJ and Eclipse can help for example in consistently updating the code base when a rename method/class/variable refactoring is performed, thus lowering the chances of introducing bugs and/or breaking tests. Given the results of our study, showing that some of these refactorings can indeed break tests, we investigate whether those breaks could have been avoided by performing these refactorings through the IDE.

Approach. We selected two popular IDEs, InteliJ and Eclipse[3], which account for 89% of the market share as of 2020 [43]. Next, we selected 135 test methods in which test breaks happened in RQ_1. Then, we cloned the repositories in which these refactorings have been performed and checked out the revision right before the refactoring implementation. We reproduced the refactoring in the production code by automating the process using the two IDEs. Finally, we evaluated whether test suites were correctly modified in consequence of the refactoring by comparing with the obtained tests with those that have been fixed by developers.

Results. Table III shows what percentage of test breaks prevented by IDEs.

TABLE III
PERCENTAGE OF THE TEST BREAKS PREVENTED BY IDES

Refactoring Type	COMPILER ERROR	FAILURE
MOVE CLASS	100% (41/41)	-
RENAME ATTRIBUTE	100% (33/33)	-
RENAME METHOD	100% (25/25)	-
ADD PARAMETER	**0% (0/13)**	**0% (0/ 2)**
RENAME CLASS	100% (10/10)	-
CHANGE RETURN TYPE	**0% (0/ 7)**	**0% (0/ 1)**
MOVE AND RENAME CLASS	100% (2/ 2)	-
CHANGE PARAMETER TYPE	**0% (0/ 1)**	-

The two selected IDEs can support all the refactoring operations that resulted in breaks in RQ_1. Both IDEs managed to avoid all the test breaks triggered by refactoring that involves the renaming or moving of code entities. Instead, all the test breaks caused by ADD PARAMETER, CHANGE RETURN TYPE and CHANGE PARAMETER TYPE refactoring are still observed even by automating the refactoring with the IDEs.

Finding 6. IDEs do not help to avoid test breaks when the methods in production code are subject to ADD PARAMETER refactoring, since an additional input is required by the refactored method, triggering changes in the test code that must be manually implemented. When an ADD PARAMETER is performed on a method, the two IDEs ask the user to provide as input the type and name of the new parameter. The user can assign a default value for the new parameter so that IDEs insert them in the invocations to the refactored method. However, all the test breaks (i.e., 13 *Compiler-Errors* and 2 *Failures*) triggered by ADD PARAMETER were still observed in our study. One major reason for such a finding is that different values for the added parameter are often required by different locations of code invoking the refactored method. Even the same test method invoking multiple times a method in the production code may require different parameters' values for the multiple invocations. However, both IDEs only allow to input one predefined value. Snippets 7 and 8 show the difference between changes required in the production and test code as a consequence of an ADD PARAMETER.

Snippet 7. Changes triggered by Add Parameter in the production code

```
  public FileVisitResult visitFile(final Path file, final
    BasicFileAttributes attributes) throws IOException {
- if (Files.exists(file) &&
-         pathFilter.accept(file)
-                 == FileVisitResult.CONTINUE) {
+ if (Files.exists(file) &&
+         pathFilter.accept(file, attributes)
+                 == FileVisitResult.CONTINUE) {
```

[3]We used InteliJ 2021.1 and Eclipse 2021-03.

Snippet 8. Example of changes triggered by Add Parameter in the test code

```
 @Test
 public void testDeprecatedWildcard() throws Exception {
 ...
-  assertEquals(FileVisitResult.CONTINUE,
-                      listFilter.accept(txtPath));
+  assertEquals(FileVisitResult.CONTINUE,
+                      listFilter.accept(txtPath, null));
```

In this example, we applied ADD PARAMETER refactoring on the method `accept` and set the default value of the newly added parameter to the variable "`attributes`". Consequently, the IDE appended the value "`attributes`" to the parameter list of all the occurrences of `accept`. However, as this variable was never declared in the test case (Snippet 8), an error occurred. Therefore, we had to manually inspect this case and choose an appropriate value. If we wanted to automate this process, the IDE should consider the context of each refactoring. Modern IDEs are still incapable of performing such a task and more efforts can be devoted into this direction.

Finding 7. When CHANGE RETURN TYPE or CHANGE PARAMETER TYPE is performed, the IDEs are unable to examine whether the impacted instance still has valid callees or has a compatible data type as a parameter, thus not helping to avoid test breaks. While the two IDEs support the CHANGE RETURN TYPE refactoring, seven *Compiler-Errors* and one *Failure* have been observed when we reproduced the refactorings causing breaking tests in RQ_1. Although the IDEs modified some of the lines that include the caller or callee of the refactored method, some lines had to be manually modified to solve *Compiler-Errors* and *Failures*. Snippet 9 shows the test code immediately after a CHANGE RETURN TYPE is performed with the IDE.

Snippet 9. Test method after a CHANGE RETURN TYPE applied with IDE

```
- final PathCounts pathCounts =
- PathUtils.deleteFile(tempDirectory.resolve(fileName));
+ final Counters.PathCounters pathCounts =
+ PathUtils.deleteFile(tempDirectory.resolve(fileName));
  Assertions.assertEquals(0,
                      pathCounts.getDirectoryCount());
  Assertions.assertEquals(1, pathCounts.getFileCount());
  Assertions.assertEquals(0, pathCounts.getByteCount());
```

Snippet 10. Example of refactoring CHANGE PARAMETER TYPE

```
- public String asString(Document doc, Properties
  properties) {
+ public static String asString(Document doc, Map<String
  , String> properties) {
    ...
   transformer.setOutputProperties(properties);
```

The IDE modifies the lines in the test code (i.e., line 1 and 2). However, the test code has compilation errors in line 5-8 because the class after refactoring (i.e., `PathCounters`) does not implement the three methods that `PathCounts` has.

As for the other cases, the IDEs do not help to avoid test breaks involving CHANGE RETURN TYPE from specific types to `void`, and any return type using generics.

While the IDEs might avoid test breaks involving several class types that can be applied by autoboxing (e.g., int to Integer), we found none of these cases in our study.

Similar scenarios also apply to CHANGE PARAMETER TYPE refactoring, which is illustrated in Snippet 10. In this example, we applied CHANGE PARAMETER TYPE refactoring on the method `asString` to modify the type of parameter "`properties`" from "`Properties`" to "`Map<String, String>`", which is incompatible with the method "`setOutputProperties`". As the IDEs are not able to detect or fix this issue, a *Compiler-Error* occurred.

Our analysis indicates that IDEs can help to avoid test breaks caused by specific refactoring operations. For our future work, we plan to assess whether these test breaks can be solved through automatic test-suite repair techniques [44][45].

Additional Analysis: IDEs are helpful to avoid test breaks triggered by refactoring involving rename or move, but they do not prevent test breaks triggered by CHANGE RETURN TYPE, CHANGE PARAMETER TYPE, or ADD PARAMETER.

B. Summary and Implications

This study examined the impact of refactoring on test suites, measuring the broken tests and fixed lines in consequence of refactoring. Additionally, this study investigated whether modern IDEs can prevent test breaks caused by different refactoring types. In this section, we map the findings across RQ_1, RQ_2, and the additional analysis performed with IDEs, and then discuss which refactoring requires more effort to fix test suites when developers refactor production code.

RQ_1 and RQ_2 showed that RENAME METHOD, MOVE METHOD and RENAME CLASS are likely to induce *Compiler-Errors* (Finding 2). Also, RENAME METHOD is the third in terms of magnitude of the requried fix in the test code (Finding 5). However, our additional analysis showed that refactoring tools in the IDE can prevent breaking test code by modifying the test code according to the change in the production code. Still, many developers conduct such refactorings manually [1][3][46], being more prone to break tests.

Take away for practitioners: When possible, the automated refactoring feature in the IDE must be exploited to minimize the chance of introducing breaking changes (and possibly bugs). This is especially true when performing refactorings such as RENAME METHOD, MOVE METHOD or RENAME CLASS, for which modern IDEs provide an excellent support.

As for ADD PARAMETER, CHANGE PARAMETER TYPE, and CHANGE RETURN TYPE, these refactoring types change the "communication interface" of the impacted methods (i.e., their input/output), which often induces *Compiler-Errors* but rarely *Failures* (Findings 2 and 4). In addition, IDEs are of little help in avoiding breaking changes when it comes to the automation of these refactoring types (Findings 6 and 7).

Thus, developers must manually select the correct argument or properly handle the new return type and fix the code location invoking the refactored method since compilers tell only the locations of the errors. Again, previous studies show that these refactorings are mostly conducted manually [1][3][46].

In addition, these refactoring types require relatively larger fixes than the other types (Finding 5) and these fixes are not trivial to perform since, for example, the developer has to select the appropriate values for the additional argument in the different locations in which the method is invoked. Thus, these types of refactoring requires a high effort for developers to fix the broken tests. Researchers could explore approaches to avoid test breaks when these types of refactoring are performed. For example, it could be helpful for developers to recommend the most appropriate input values in the invocations of the refactored method. This could be done, by comparing trace logs generated by various inputs for the method refactored with ADD PARAMETERS with the original trace logs (i.e., before refactoring).

Take away for researchers: Researchers and IDE developers should improve the support provided when implementing ADD PARAMETER, CHANGE PARAMETER TYPE, and CHANGE RETURN TYPE refactoring, especially when these refactoring impact multiple locations in the test/production code.

C. Threats to Validity

Internal Validity. Internal validity threats concern factors internal to our study that could influence our results. To investigate the inherent impact of refactoring, we decided to exclude from our study test methods that were affected by multiple refactoring edits. For example, in one commit we found a single test method that exercised the production code locations impacted by 19 refactoring edits. Our choice was driven by the will of isolating to the extent possible the impact of the different refactoring types. However, we acknowledge that our analysis may miss interesting test breaking cases resulting from combinations of refactoring.

Construct Threats. Construct validity threats concern the relationship between theory and observation, and are mainly related to sources of imprecision in our analyses. We employed dynamic analysis in an effort to establish precise links between production and test code.

Compared with FaultTracer [47] used in the previous work [8], our approach can detect line-level refactoring. However, it might produce more false positives. Thus, we manually validated the data used in our study to increase the confidence in our findings. As for manual inspection, to avoid subjectiveness bias, all the manual analyses we performed were run by two authors, and a third author was involved in case of conflicts. Still, we cannot exclude imprecisions in such a process.

External Threats. External validity threats concern the generalizability of our findings. We selected eight projects for our quantitative study, paying attention to select projects having a different size and belonging to different application domains. The number of projects is comparable with that of similar previous studies [8][21].

Still, there are many factors negatively impacting the generalizability of our findings. First, we only focused on Java systems using Maven. Second, we only employed projects that do not include in their repository any sub-project to avoid dependency issues. Third, the number of projects (8) is clearly too low to claim any generalizability of our findings. However, we want to highlight the costly analysis performed to collect the data used in our experiments.

VI. CONCLUSIONS AND FUTURE WORK

This study measured the impact of refactoring on test suites. We conducted a large-scale and fine-grained study using dynamic analysis and manual inspection of the collected data. We employed RefactoringMiner, the state-of-the-art refactoring detection tool to identify refactoring edits performed in eight open source projects. We checked whether test methods exercise location in the production code subject to refactoring-related edits. This was done by collecting and parsing execution log traces. Although collecting this data on every commit requires a formidable amount of time, we overcame this issue by expoloiting Kubernetes and numerous docker instances.

Throughout our empirical study we ran more than 615,196 test methods, and we observed that most types of refactoring operations do not break test suites and, even in the rare occasions when they do, small fixes are required to the related tests. A few refactoring operations are more likely to break test suites. For example, ADD PARAMETER results in compilation errors for 6.7% of test methods and failures for 1.0% of them. Also, several lines of code must usually be fixed in test methods as a result of this refactoring. Finally, we observed that for refactoring renaming and moving code entities, exploiting the automated support of IDEs can help in avoiding test breaks. Instead, this is not the case for ADD PARAMETER, CHANGE PARAMETER TYPE, and CHANGE RETURN TYPE refactoring.

Overall, based on the evidence we found through our study, the question of whether refactoring breaks tests is to be answered with a sound "no, it does not, and it should not". It does not, as by definition refactoring is a behavior-preserving code transformation process, thus contradicting previous findings by Rachatasumrit and Kim [8], who however did not heed the issue of tangled commits. We phrased this finding as a take-away for practitioners. However, for refactoring operations that modify the method signatures and do induce test breaks we also found a gap in terms of IDE support, which we phrased as a take-away for researchers.

Our future work fill focus on (i) the research challenges we listed in Section V, and (ii) corroborating our findings by running our study on a larger set of systems.

ACKNOWLEDGMENT

We thank Dr. Ishio and his colleagues for developing SELogger and giving us helpful advice. We also gratefully acknowledge the financial support of JSPS and SNSF for the project "SENSOR" (No. 183587, JPJSJRP20191502), and JSPS for the KAKENHI grants (JP21H04877, JP21K17725).

REFERENCES

[1] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

[2] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 29:1–29:30, 2020.

[3] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the 20th ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012, p. 50.

[4] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. L. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca, M. Ribeiro, C. Barbosa, and D. Oliveira, "How does incomplete composite refactoring affect internal quality attributes?" in *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. 149–159.

[5] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 101–110.

[6] C. Vassallo, F. Palomba, and H. C. Gall, "Continuous refactoring in CI: A preliminary study on the perceived advantages and barriers," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 564–568.

[7] B. Adams and S. McIntosh, "Modern release engineering in a nutshell - why researchers should care," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 78–90.

[8] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 357–366.

[9] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, and A. Zaidman, "Continuous delivery practices in a large financial organization," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 519–528.

[10] K. O. Elish and M. Alshayeb, "Investigating the effect of refactoring on software testing effort," in *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC)*, 2009, pp. 29–34.

[11] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative*, 2013, pp. 132–146.

[12] F. Shull, J. C. Carver, S. Vegas, and N. J. Juzgado, "The role of replications in empirical software engineering," *Empirical Software Engineering (EMSE)*, vol. 13, no. 2, pp. 211–218, 2008.

[13] O. S. Gómez, N. J. Juzgado, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology (IST)*, vol. 56, no. 8, pp. 1033–1048, 2014.

[14] B. Cornelissen, L. Moonen, and A. Zaidman, "An assessment methodology for trace reduction techniques," in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008, pp. 107–116.

[15] Kubernetes. https://kubernetes.io/.

[16] X. Ren, B. G. Ryder, M. Störzer, and F. Tip, "Chianti: a change impact analysis tool for java programs," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 664–665.

[17] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," in *Proceedings of the 16th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2001, pp. 312–326.

[18] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *Proceedings of the 16th ACM International Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 71–82.

[19] A. K. Onoma, W. Tsai, M. H. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[20] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002, pp. 97–106.

[21] X. Tang, S. Wang, and K. Mao, "Will this bug-fixing change break regression testing?" in *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 215–224.

[22] M. Bruntink and A. van Deursen, "An empirical study into class testability," *The Journal of Systems and Software (JSS)*, vol. 79, no. 9, pp. 1219–1232, 2006.

[23] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality (WoSQ)*, 2007, p. 10.

[24] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: A multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, 2017, pp. 74–83.

[25] 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 (ESEC/FSE)*, 2017, pp. 465–475.

[26] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 95–104.

[27] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Proceedings of the 2008 Balancing Agility and Formalism in Software Engineering*, 2008, pp. 252–266.

[28] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 104–113.

[29] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding the best compromise between design quality and testing effort during refactoring," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 24–35.

[30] A. Sabane, M. D. Penta, G. Antoniol, and Y. Guéhéneuc, "A study on the relation between antipatterns and the cost of class unit testing," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 167–176.

[31] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[32] JUnit. https://junit.org/.

[33] TestNG. https://testng.org/.

[34] J. Farley and W. Crawford, *Java Enterprise in a Nutshell*. O' Reilly & Associates, Inc., 2005.

[35] Maven. https://maven.apache.org/.

[36] G. Soares, R. Gheyi, E. R. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *The Journal of Systems and Software (JSS)*, vol. 86, no. 4, pp. 1006–1022, 2013.

[37] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: should we consider branches?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 202–213.

[38] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, "Near-omniscient debugging for java using size-limited execution trace," in *Proceeding of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 398–401.

[39] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, pp. 159–174, 1977.

[40] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?" *Empirical Software Engineering (EMSE)*, vol. 25, no. 1, pp. 790–823, 2020.

[41] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 643–653.

[42] Oracle. https://docs.oracle.com/javase/tutorial/java/javaoo/methods.html.

[43] CODEAHOY. https://codeahoy.com/java/top-5-java-ides/.

[44] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "Reassert: a tool for repairing broken unit tests," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1010–1012.

[45] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 217–227.

[46] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[47] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: a change impact and regression fault analysis tool for evolving java programs," in *Proceedings of the 20th ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012, p. 40.