Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Test Code Refactoring Unveiled: Where and How Does It Affect Test Code Quality and Effectiveness?

Luana Martins · Valeria Pontillo · Heitor Costa · Filomena Ferrucci · Fabio Palomba · Ivan Machado

Received: date / Accepted: date

Abstract Refactoring has been widely investigated in the past in relation to production code quality, yet little is known about how developers apply refactoring to test code. Specifically, there is still a lack of investigation into how developers typically refactor test code and its effects on test code quality and effectiveness. This paper presents an exploratory empirical study aimed to bridge this gap of knowledge by investigating (1) whether test refactoring actually targets test classes affected by quality and effectiveness concerns and (2) the extent to which refactoring contributes to the improvement of test code quality and effectiveness. First, we performed an exploratory mining software repository to collect test refactoring data of open-source Java projects from GitHub. Then, we statistically analyzed them in combination with quality metrics, test smells, and code/mutation coverage indicators. Furthermore, we measured how refactoring operations impact the quality and ef-

Luana Martins
Federal University of Bahia, Salvador, Brazil
E-mail: martins.luana@ufba.br

Valeria Pontillo
Software Languages (Soft) Lab — Vrije Universiteit Brussel, Brussels, Belgium
E-mail: valeria.pontillo@vub.be

Heitor Costa
Federal University of Lavras, Lavras, Brazil
E-mail: heitor@ufla.br

Filomena Ferrucci
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: fferrucci@unisa.it

Fabio Palomba
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: fpalomba@unisa.it

Ivan Machado
Federal University of Bahia, Salvador, Brazil
E-mail: ivan.machado@ufba.br

fectiveness of test code. Our findings indicate that test refactorings primarily address low-quality test code, as evidenced by test smells and quality metrics. At the same time, we did not find a statistically significant relationship between test refactorings and code/mutation coverage metrics. In addition, test refactorings enhance the coupling, cohesion, and size of the test code, albeit sometimes leading to an increase in certain test smells. We conclude our study by emphasizing the significance of incorporating both quality metrics and test smells into refactoring decisions to enhance the overall quality of test code.

Keywords Software Testing · Test Code Quality · Test Refactoring · Empirical Software Engineering.

1 Introduction

Refactoring is an engineered approach that allows developers to improve source code quality without affecting its external behavior [18]. Over the last decades, researchers have been proposing automated refactoring recommenders [7] and investigated how refactoring relates to code quality [4,53,16,10]. In particular, researchers identified both benefits and drawbacks of its application [1,5,27], finding that, while refactoring is theoretically associated with modifications that do not affect the external behavior of source code, it may induce defects [15,6,17], vulnerabilities [22], or even code smells [59]. These drawbacks are mainly due to refactoring activities performed manually without the support of automated tools and interleaved with other code changes [37]. Our research is motivated by these previous works. On the one hand, most previous studies focused on the refactoring of production code and, for this reason, we argue that there is a lack of investigations into *how refactoring is applied to test code*. On the other hand, it remains unclear whether similar effects observed in previous studies may emerge with test refactoring, particularly regarding its potential impact on both test quality and effectiveness. For instance, when refactoring actions target the logic of a test case, there may be repercussions on both aspects. Hence, we point out a *limited knowledge on the effects of refactoring* on both test quality and effectiveness.

An improved understanding of test refactoring would have a number of potential benefits for research and practice. In the first place, test cases represent a crucial asset for software dependability: developer's productivity is partly dependent on the quality of test cases [36], as these help practitioners to decide on whether to merge pull requests or deploy the system [20]. As such, analyzing how refactoring affects test cases may have a significant impact on practice. Secondly, researchers have been showing that the design of test code is approached in a substantially different way with respect to traditional development [34]. Indeed, the test code must often interact with external systems, databases, or APIs to set up test environments and verify the system's behavior [35]. As a consequence, test code may suffer from different issues that, in turn, would require different refactoring operations [21].

For these reasons, new refactoring practices have been proposed with the aim of dealing with quality or effectiveness concerns [13,35,21]. While those refactoring practices were the target of some previous investigations, researchers limited their focus to how refactoring may influence test smells, i.e., symptoms of poor test code quality [51,52,46], hence not providing comprehensive analysis into the *nature* and *effects* of test refactoring. More specifically, we highlight a lack of knowledge on (1) whether developers apply test refactoring operations on test classes that are actually affected by quality or effectiveness concerns, as it is supposed to be based on the definition of refactoring; and (2) what is the effect of refactoring on both quality and effectiveness of test cases.

This paper aims to address this gap of knowledge by conducting an *exploratory empirical study*.[1] We first collect test refactoring data from the change history of open-source Java projects from GitHub and combine them with data coming from automated instruments able to profile test code from the perspective of quality metrics, test smells, and code/mutation coverage information. Afterward, we apply statistical analyses to address three main research goals targeting (1) whether test classes with a low level of quality, in terms of test smells and code metrics, are associated with more test refactoring, (2) whether a low level of effectiveness, in terms of mutation coverage and code coverage, is associated with more test refactoring, and (3) to what extent the removal of test smells improve the test code quality and effectiveness. As such, the scientific novelty of our article lies in two key aspects. In the first place, our research focuses specifically on *test refactoring operations*, which is, to the best of our knowledge, still unexplored. Even if test refactoring shares underlying principles with traditional refactoring, the different nature of test code requires distinct refactoring approaches and may lead to different outcomes compared to production code. Second, we broaden our analysis by not only examining the correlation between test refactoring and test code quality attributes, but also investigating its impact on key test code effectiveness indicators, such as branch and mutation coverage. This deeper exploration enhances the current understanding of the factors that influence these critical metrics, thereby expanding the body of knowledge in this domain.

Our main findings show that test refactorings target low-quality test code regarding test smells and quality metrics. Still, there is no strong indication that coverage and mutation scores drive the refactorings under investigation. In addition, refactorings from Fowler's catalog, e.g., *Extract Class*, improve the coupling, cohesion, and size of the test code. In contrast, test-specific refactoring may lead to an increase in the number of test smells.

Our findings might benefit researchers and practitioners from multiple perspectives. In the first place, our research may reveal insights into the refac-

[1] This manuscript represents the complete version of our previous registered report [32]. Compared to our initial plan, we reduced the sample size from 175 to 100 projects due to challenges encountered during the execution of our tooling. Specifically, the need to compile historical snapshots for computing independent variables introduced various technical limitations, such as compilation issues and dependency resolution problems, which limited the number of projects we were able to analyze, as discussed in detail in Section 4.

toring types that may deteriorate test code quality and effectiveness. Such information would be relevant for researchers in both the fields of refactoring and testing, as it may lead them to (1) extend the knowledge on the best and bad practices to properly apply test refactoring; (2) devise novel test refactoring approaches which are aware of the possible side effects of refactoring, e.g., we may envision multi-objective search-based refactoring approaches that may optimize refactoring recommendations based on both quality and effectiveness attributes; and (3) design novel recommendation systems that may support developers in understanding how a refactoring would impact different test code properties. The results would also be useful to practitioners, who may have additional proof of the side effects of refactoring, hence possibly being stimulated further on the need to employ automated refactoring tools. In the second place, our findings may indicate the nature of the test cases more likely to be subject to refactoring operations. Researchers might use this information to define refactoring recommenders and refactoring prioritization approaches, while practitioners may acquire awareness of their actions.

Structure of the paper. Section 2 overviews the most closely related work, positioning our work within the current body of knowledge. Section 3 elaborated on the research questions of the study, while Section 4 reports on the research methods employed to address them. In Section 5, we analyze the results achieved from our analyses. Section 6 further discusses the main findings of our work, emphasizing the implications for research and practice. The potential limitations of our study are discussed in Section 7. Finally, Section 8 concludes the paper and outlines our future research agenda.

2 Related Work

The current literature can be distinguished based on the type of empirical studies conducted. First, several studies analyzed change history information to extract knowledge about test smells and their impact. Spadini et al. [54] investigated ten open-source projects to find a relation between six test smells and the change and defect-proneness of both test and production code, finding that smelly JUnit tests are more change-prone and defect-prone than non-smelly ones. In addition, they found that production code is typically more defect-prone when tested by smelly tests. As such, the authors did not target test code refactoring, hence not assessing how the seemingly test code quality improvement actions performed by developers affect test code quality and effectiveness, i.e., the authors looked exactly in the opposite direction of our paper, focusing on how bad practices affect test code quality.

Wu et al. [64] explored the impact of eliminating test smells on the production code quality of ten open-source projects. In this respect, there are two key points that make our investigation novel: first, test smell removal does not imply the application of refactoring. A previous empirical study [23] indeed showed that 83% of test smell removal activities are due to feature maintenance actions. Hence, our work can therefore further the knowledge of how

developers apply test code refactoring. Second, the authors focused on the effects of test smells on code quality rather than analyzing the impact of test code refactoring actions. As such, our work extends the current knowledge by assessing how test refactoring is applied and the impact on multiple aspects of test code, such as quality and effectiveness.

Peruma et al. [46] investigated the relationship between refactoring changes and their effect on test smells. The authors used RefactoringMiner [57] to detect refactoring operations and the tsDetect tool [45] to identify the test smells from unit test files of 250 open-source Android Apps. Results showed that refactoring operations in test and non-test files differ, and the refactorings co-occur with test smells. With respect to the work by Peruma et al. [45], we do not limit ourselves to the analysis of test smells, but also consider additional indicators of test code quality and effectiveness: in this sense, ours represent a more comprehensive analysis of the role of test refactoring. Second, we assess the actual effects of test refactoring on test code quality and effectiveness, providing insights into how various test refactoring types may support the evolutionary activities of developers.

A second line of research is represented by qualitative studies targeting the developer's perception of test refactoring. Damasceno et al. [12] investigated the impact of test smell refactoring on internal quality attributes, reporting some insights that may potentially be in line with the results of our study, e.g., they let emerge the impact of test smell refactoring on internal quality attributes. In the first place, the authors focused on the refactoring of test smells, while our work targets test code refactoring from a more general perspective, attempting to assess the extent to which this is applied to classes, suggesting the presence of quality or effectiveness concerns. Secondly, our results may provide evidence-based, complementary insights with respect to what the authors found out in their qualitative study. Third, our work has a broader scope, and it also targets the effectiveness side of the problem.

Soares et al. [51] investigated how developers refactor test code to remove test smells. The authors surveyed 73 open-source developers and submitted 50 pull requests to assess developers' preferences and motivation while refactoring the test code. The results showed that developers preferred the refactored test code for most test smells. In another work, Soares et al. [52] investigated whether the JUnit 5 features help refactor test code to remove test smells. They conducted a mixed-method study to analyze the usage of the testing framework features in 485 popular Java open-source projects, identifying the features helpful for test smell removal and proposing novel refactorings to fix test smells. Also in this case, the authors focused on the refactoring of test smells, while our study has a broader scope. In addition, while we did not conduct surveys or interviews—this is part of our future research agenda—we extended the current body of knowledge by assessing whether test code quality and effectiveness indicators may trigger refactoring activities, other than providing a comprehensive overview of how test refactoring relates to branch and mutation coverage, which is a premiere of our study.

3 Research Questions and Objectives

The *goal* of the empirical study was to analyze the test refactoring operations performed by developers over the history of software projects, with the *purpose* of understanding (1) whether low-quality test classes, in terms of structural metrics and test smells, provide indications on which test classes are more likely of being refactored, (2) whether test classes with low effectiveness, in terms of code coverage and mutation coverage, provide indications on which test classes are more likely of being refactored, and (3) as a consequence, to what extent test refactoring operations are effective in improving quality and effectiveness of test classes. In other terms, we were first interested in assessing the **quantity** of test refactoring operations performed on classes exhibiting test code quality and effectiveness issues. Then, we assessed the **quality** of the test refactoring operations applied in terms of improvements provided to test code quality and effectiveness. The *perspective* is of both researchers and practitioners who are interested in understanding the relationship and effects of test refactoring operations on the quality and effectiveness of test classes.

More specifically, our empirical investigation aims to address first the following research questions (**RQ**s):

RQ$_1$. *Are test refactoring operations performed on test classes having a low level of quality, as indicated by quality metrics and test smell detectors?*

RQ$_2$. *Are test refactoring operations performed on test classes having a low level of effectiveness, as indicated by code and mutation coverage?*

Through **RQ**$_1$ and **RQ**$_2$, we aim to address the first objective of the study, hence understanding whether the low quality and effectiveness of test classes are associated with more test refactoring operations. The results of these two research questions might have multiple implications for software maintenance, evolution, and testing researchers. An improved understanding of these aspects may inform researchers on the characteristics of test suites that trigger more refactoring operations, possibly providing insights on (1) the factors associated with test refactoring and (2) the design of novel or improved instruments to better support developers. For instance, should we discover that test refactoring is not frequently applied on test classes exhibiting test smells, this would suggest the need for further research into the motivation leading developers to refactor test code. Moreover, it would also point to the need for better design of test smell detectors to facilitate refactoring efforts.

Upon completion of this investigation, we further elaborated on the impact of test refactoring, addressing the following research questions:

RQ$_3$. *What is the effect of test refactoring on test code quality, as indicated by quality metrics and test smell detectors?*

RQ$_4$. *What is the effect of test refactoring on test code effectiveness, as indicated by code and mutation coverage?*

Through $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$, we aim to extend the current knowledge on the impact of test refactoring, assessing whether the test code quality and effectiveness increase, decrease, or remain the same after the application of test refactoring operations. It is worth mentioning that addressing these two research questions would be important independently from the results obtained by $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. Indeed, regardless of the amount of refactoring operations performed on test classes exhibiting quality or effectiveness concerns, it would still be possible that the specific refactoring actions targeting those classes have an impact. To make our argumentation more practical, consider the case of the *Extract Method* refactoring, whose suboptimal implementation may potentially affect test code effectiveness. Given a verbose test method with several steps and assertions, the refactoring enables the extraction of multiple test methods, which are supposed to be more cohesive and focused on the verification of specific conditions of production methods. However, if the extraction is not performed properly, it could alter the logic of the test and negatively impact test effectiveness. For instance, consider test T, which verifies two branches, B1 and B2, of the production method M. In this case, an Extract Method operation is supposed to split T so that the resulting tests T1 and T2 target B1 and B2 individually. However, should there be a logical relation between B1 and B2, T2 will still need to pass through T1 to ensure that the logical relation is still met. A suboptimal refactoring may overlook this requirement, possibly not embedding in T2 the statements required to reach B1. As a result, this operation would affect the overall level of coverage of the production code.

As such, $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$ provide an orthogonal view on the matter. Also in this case, the outcome of our investigation may lead to implications for research and practice. First, our findings may help researchers measure the actual, practical impact of test refactoring—this may drive considerations on how future research efforts should be prioritized, e.g., by favoring more research on impacting refactoring operations. Second, our results may increase the practitioner's awareness of test refactoring, possibly increasing its application in practice.

To design and report our empirical study, we will follow the empirical software engineering guidelines by Wohlin et al. [63] as well as the ACM/SIGSOFT Empirical Standards.[2]

4 Experimental Plan

This section reports the research method that we applied to address our **RQs**. Figure 1 overviews the main steps conducted to execute our study.

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

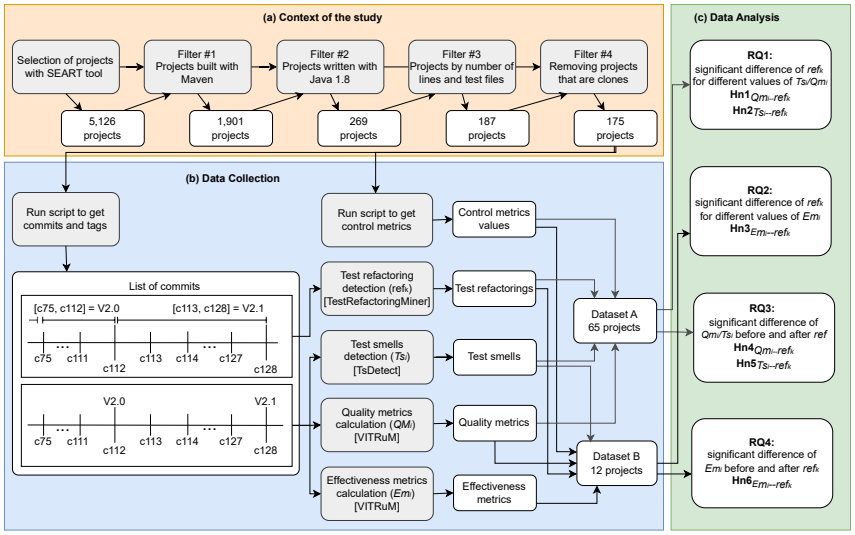


Fig. 1: Overview of the experimental design.

4.1 Context of the study

The *context* of our investigation was composed of (i) software systems, i.e., the projects that have been mined to collect the data required to address our research objectives, and (ii) empirical study variables, i.e., the independent and dependent variables that we statistically analyzed.

Software Systems. The selection of suitable software systems was driven by various considerations. First, we focused on open-source projects, as we needed access to information on change history. Second, we relied on popular, large real-world projects with enough releases to collect data that could be analyzed statistically. Third, we standardized the building process to ease dependency management and streamline build configurations across all projects. As such, we used SEART tool[3] to select open-source, non-fork projects from GitHub that had at least 100 stars, 10 major releases, 1,000 lines of code, and 10 test classes. We sought Java projects that can be compiled with Maven and Java 8—Java 8 is the most popular version used at the time of writing.[4] Figure 1 (a) shows the number of projects identified after applying each selection criterion. We ultimately identified 175 potentially suitable projects, which form the sample considered for the analysis.

The selected projects vary in terms of scope, size, and communities - more details about the characteristics of the projects are reported in our online appendix [33]. In addition, we thoroughly reviewed the contribution guidelines

[3] `https://seart-ghs.si.usi.ch/`

[4] `https://www.jetbrains.com/lp/devecosystem-2023/java/`

of the projects to assess whether any of them followed the so-called *Boy Scout* rule, i.e., *"Leave every piece of code you touch cleaner than how they found it"*.[5] We control for this factor since projects following this principle might be more inclined to the application of refactoring. Therefore, they could have higher test code quality and effectiveness, possibly diverging from standard behaviors. We did not find explicit references to this principle in any of the selected projects. Yet, this practice may still be embedded in their development processes because of developers' commitment to clean code practices, even if not formally documented in the guidelines. For example, within the Adobe-Consulting-Services/acs-aem-commons project, one contributor explicitly adheres to this principle, as evidenced by pull requests.[6] We extended such an investigation by inquiring the core maintainers of the projects considered, asking whether they consistently apply this rule or are aware of contributors applying it. Unfortunately, we did not receive any feedback from these maintainers, yet our manual investigation suggests that the practice is not significantly widespread and this factor may not bias our analysis.

Moreover, we characterized the projects based on the automated static analysis tools used in their quality assurance processes. Developers might be inclined to follow the feedback from these tools, possibly making them more prone to refactor test code. Although 37 projects reference SonarQube, Checkstyle, or PMD in their configuration files, only 12 projects explicitly ask contributors to fix the warnings raised by these tools before committing new code. However, these projects were excluded from the study due to compilation issues encountered during data collection. In conclusion, we provide two observations. First, the explicit fixing of warnings raised by static analysis tools is not systematically requested by the core contributors of the open-source projects in our sample, though some developers may be more committed to clean code practices. Second, none of the static analysis tools target the test code quality concerns considered in our study nor suggest the refactoring operations we investigated. Thus, it is unlikely that the use of these tools has biased our findings.

Empirical Study Variables. In the context of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we were interested in assessing whether refactoring operations were more likely to be observed on test classes exhibiting test code quality and effectiveness concerns. As such, we defined the following empirical study variables:

Independent Variables. These are the factors that are related to the application of test refactoring, namely (i) test code quality metrics; (ii) presence of test smells (of different types); (iii) branch coverage; and (iv) mutation coverage. Tables 1 and 2 list and describe the independent variables of the study. These metrics were all computed across releases of different software systems and were statistically analyzed, as described later. The selection of

[5] The Boy Scout Rule: `https://www.oreilly.com/library/view/97-things-every/9780596809515/ch08.html`.

[6] Available at: `https://github.com/Adobe-Consulting-Services/acs-aem-commons/pull/1908`

Table 1: Description of quality metrics as detected by VITRuM. The description includes an interpretation of the metrics, along with reference mean values identified in the referred previous studies. These values were derived by these previous studies from analyses of metric distributions in high-quality projects, providing average benchmarks that can be considered acceptable standards for the interpretation of our results.

Quality Metrics	**Description**
Number of Lines (LOC)	Counts the number of lines of code. The higher the metric values, the higher the size of the class. Oliveira et al. [40] suggested that classes having a $LOC \geq 222$ are more prone to be defective.
Number of Methods (NOM)	Counts the number of methods. High metric values indicate that the classes have more responsibilities. Oliveira et al. [40] suggested that classes having a $NOM \geq 16$ are harder to maintain.
Weight Method Class (WMC)	Counts the number of branch instructions in a class. The higher the metric values, the more complex the class. Shatnawi et al. [49] suggested that classes having a $WMC \geq 32$ are more fault-prone.
Response for a Class (RFC)	Counts the number of method invocations in a class. High metric values indicate that a class interacts with many other classes, leading to increased complexity. Shatnawi et al. [49] suggested that classes having a $RFC \geq 49$ are more fault-prone.
Assertion density (AsD)	Calculates the percentage of asserts with respect to the total number of statements in a class. A greater number of assertions suggests a more thorough testing regime. Kudrjavets et al. [25] suggested that 34 assertions per KLOC, i.e., $AsD \leq 3.4\%$, indicate that tests cover a reasonable amount of behaviors without being overly specific.
Mutation Coverage (MT)	Calculates the percentage of mutated statements in the production class covered by the test. A higher MT indicates better test coverage. Schweikl et al. [48] observed that many projects have a coverage of around $MT \geq 34\%$.
Line coverage (LC)	Calculates the percentage of lines exercised by the test. A higher LC indicates better test coverage. Schweikl et al. [48] observed that many projects have $LC \geq 62\%$.
Branch coverage (BC)	Calculates the percentage of branches exercised by the test. Schweikl et al. [48] observed that many projects have a coverage of around $BC \geq 56\%$.

Note: The reported values should be considered as guidelines for interpreting our findings rather than universally established standards, as the ideal range for these metrics can vary depending on the project context, language, and domain.

these independent variables was driven by multiple considerations. First, we consider test code quality metrics and test smells that were targeted by previous research in the field [9,44] and found to impact test code in different manners [55,23]. Second, branch and mutation coverage are widely considered two key indicators of test code effectiveness, which may estimate the goodness of test cases in dealing with real defects [24,42].

Dependent Variables. These are the refactoring operations (of different types) being observed across releases of different software systems. To select suitable test refactoring operations for our purpose, we investigated the literature

discussed in Section 2 to extract the test refactoring operations previously associated with our independent variable. Table 3 lists the refactoring operations that have been targeted, along with a brief description.

Table 2: Description of test smells as detected by tsDetect.

Test Smell	Description	Precision	Recall
Assertion Roulette (AR)	A test method contains assertion statements without an explanation/message	94.7%	90.0%
Duplicate Assert (DA)	A test method that contains more than one assertion statement with the same parameters	85.7%	90.0%
Handling Exception (ECT)	A test method that contains throws statements	100.0%	100.0%
Eager Test (ET)	A test method contains multiple calls to multiple production methods	100.0%	100.0%
General Fixture (GF)	Fields within the setUp method are not utilized by all test methods	95.2%	100.0%
Lazy Test (LT)	Multiple test methods call the same class under test methods	90.9%	100.0%

When it turned to $\mathbf{RQ}_3$ and $\mathbf{RQ}_4$, we were interested in assessing the impact of test refactoring on the test code quality and effectiveness aspects considered. As such, we needed to swap independent and dependent variables: indeed, in this case, we were interested in observing how refactoring impacted test code properties rather than the opposite:

Independent Variables. These are the different refactoring operations reported in Table 3, computed across the releases of software systems considered.

Dependent Variables. These are the test code quality and effectiveness metrics described in Tables 1 and 2, which have been computed across releases of software systems.

In all **RQ**s, we also included a set of control variables to analyze whether project- and process-level characteristics may affect the dependent variables.

Control Variables. We took into account the frequency of releases and activities by the project. Such information may provide insights into the development speed, which, in turn, may impact test code quality and effectiveness. Given a release R_i, we computed the number of releases issued within the last 1, 3, 6, and 12 months. In addition, for each class C_j within R_i, we computed the number of commits performed by developers between the releases R_{i-1} and R_i. We also considered project-level metrics such as (1) project size in terms of lines of code; (2) number of contributors; (3) number of branches; and (4) number of pull requests. On the one hand, these metrics can provide a good overview of the main characteristics of the project and

Table 3: Description of refactorings detected by TestRefactoringMiner.

Refactoring	Description	Precision	Recall
Add Assert Explanation	Add an optional parameter into the assert methods to provide an explanatory message	100.0%	78.0%
Extract Class	Create a new class and place the fields and methods responsible for the relevant functionality in it	100.0%	100.0%
Extract Method	Move a code fragment to a separate new method and replace the old code with a call to the method	99.9%	96.9%
Inline Method	Replace calls to the method with the method's content and delete the method itself	100.0%	98.2%
Parameterize Test	Remove duplicate code using the @parameterized test annotation to define a variety of arguments	100.0%	100.0%
Replace @Test annotation w/ assertThrows	Remove @Test annotation and add of assertThrows method	100.0%	93.0%
Replace @Rule annotation w/ assertThrows	Remove @Rule annotation and add of assertThrows method	100.0%	88.0%
Replace try/-catch w/ assertThrows	Remove try/catch blocks and add of assertThrows method	100.0%	89.0%

the community around it. On the other hand, all these metrics can impact test code quality and effectiveness, e.g., a higher amount of branches may indicate a higher level of activity around the project, which in turn can influence the way test cases are maintained and evolved.

4.2 Data Collection

We used different automated tools available in the literature to extract data on quality and effectiveness metrics, test smells, and refactoring operations. Figure 1 (b) shows the data collection and integration to compose our datasets.

Collecting test code quality and effectiveness metrics. To collect both test code quality and effectiveness metrics (Table 1), we ran VITRuM, a plug-in for the computation and visualization of static and dynamic test-related metrics [43].The tool uses libraries such as JaCoCo to calculate line and branch coverage and pitest for the mutation coverage. It is important to remark that those libraries require building the projects. Due to various compilation issues, e.g., dependency resolution [29,58], VITRuM could not analyze several projects, as further elaborated in the remainder of this section. Similar challenges have been documented in prior research on mining repositories for dynamic metrics. For instance, Tufano et al. [58] reported that

only 38% of the change history could be successfully compiled when performing mining studies, demonstrating the hardness of analyzing the evolution of software projects from the perspective of dynamic information.

Collecting test smells. Among the test smell detection tools available for Java code [2], we used tsDetect [45], which is the most accurate tool, with a precision score ranging from 85% to 100% and a recall score ranging from 90% to 100%. tsDetect performs a test code static analysis through an AST (Abstract Syntax Tree) to apply the test smells detection rules in the test files. A test file in the JUnit testing framework should follow the naming conventions of either pre-pending or appending the word ``Test'' to the name of the production class under test and at the same package hierarchy [45]. With the detection rules, the tool can detect (i) the presence or absence of a test smell in a test class or (ii) the number of instances per test smell in a test class. In addition, the tool receives a configuration of the severity thresholds for each test smell [56]. We ran the tool to identify the number of instances of the six test smells described in Table 2 with default values for the severity thresholds (i.e., the tool reports all instances of test smells detected).

Collecting refactoring data. To detect test refactoring operations, we used the TestRefactoringMiner tool [31]. The tool is built on top of the state-of-the-art refactoring mining tool RefactoringMiner, which has the highest precision (99.8%) and recall (97.6%) scores among the currently available refactoring mining tools [57]. The tool analyzes the added, deleted, and changed files between two project versions to detect specific test refactorings, reaching 100% and 92.5% of precision and recall scores. In order to retrieve all the test refactorings performed in a release, we had to group the test refactorings returned by TestRefactoringMiner for the commits ranging from $release_{n-1}$ to $release_n$. As a result, the tool operationalizes the detection of all the refactoring operations considered in the study—Table 3 presents the set of test refactoring investigated. It is worth noting that this set considers various refactoring operations, such as integrating new technologies like JUnit 5 or improving the organization of test classes.

Data integration. We established traceability links between the test classes reported by tsDetect, VITRuM, and TestRefactoringMiner tools, finally integrating their outcome in a unique data source to be further analyzed from a statistical standpoint. However, the tools did not compute the metrics for the same test classes for three main reasons. First, VITRuM and tsDetect calculate metrics for specific releases in the project's history, while TestRefactoringMiner groups refactored classes between consecutive releases and reports test refactorings. Thus, the integration of the tools' outputs depends on the number of refactored classes detected by TestRefactoringMiner, while VITRuM and tsDetect track metrics for each class over time. Second, VITRuM could not calculate test coverage and mutation scores for some projects due to issues such as dependency resolution [29,58]. Third, the tools occasionally failed due to exceeding available memory on the machine or internal issues. Consequently, some test classes have refactorings but do

not match the quality metrics, coverage, or mutation scores. As a consequence of these observations, we had to adjust our dataset construction strategy by considering two sets of projects. **Dataset A** contains static metrics extracted from 65 projects, i.e., quality metrics, test smells, and refactoring operations. **Dataset B** contains static and dynamic metrics extracted from 12 projects, i.e., quality metrics, test smells, effectiveness metrics, and refactoring operations. Please note that the 12 projects contained in Dataset B are also included in Dataset A. This refinement allowed the data analysis while acknowledging the limitations encountered in achieving our initial goal of analyzing a larger amount of systems [32]. Figure 2 shows the characterization of both datasets.

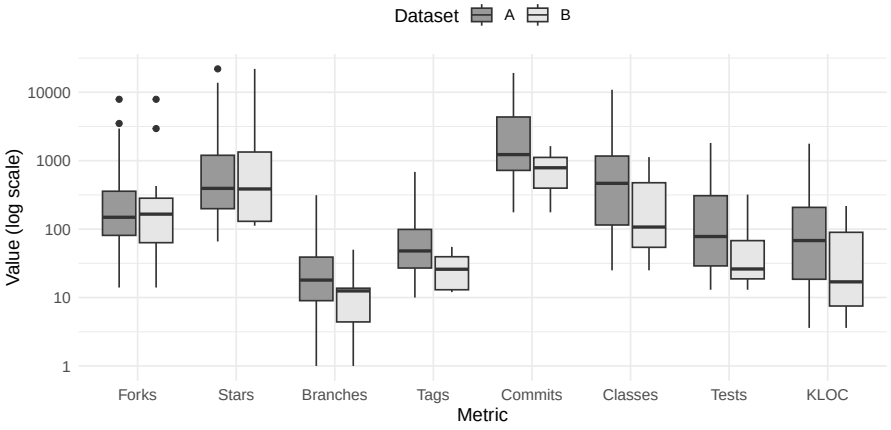


Fig. 2: Characterization of datasets (boxplots in log scale)

4.3 Data Analysis

We first formulate the working hypotheses that we statistically assessed. As for $\mathbf{RQ}_1$, given a quality metric Qm_i in the set of test code quality metrics considered in the study and a refactoring ref_k in the set of refactoring operations considered in the study, our null hypothesis was the following:

Hn1$_{Qm_i - ref_k}$**.** There is *no significant difference* in terms of the amount of ref_k operations performed on test classes having different values of Qm_i.

As in $\mathbf{RQ}_1$, we also evaluated the relation between test refactoring and test smells. Given a test smell Ts_i in the set of test smells considered in the study and ref_k, we defined a second null hypothesis:

Hn2$_{Ts_i - ref_k}$**.** There is *no significant difference* in terms of the amount of ref_k operations performed on test classes affected and not by Ts_i.

As for $\mathbf{RQ}_2$, given an effectiveness metric Em_i in the set of effectiveness metrics considered in our study and ref_k, the null hypothesis was the following:

Hn3$_{Em_i - ref_k}$**.** There is *no significant difference* in terms of the amount of ref_k operations performed on test classes having different values of Em_i.

As for $\mathbf{RQ}_3$, given a quality metric Qm_i, a test smell Ts_i, and a refactoring ref_k, the null hypotheses was:

Hn4$_{Qm_i - ref_k}$**.** There is *no significant difference* in terms of Qm_i before and after the application of ref_k.

Hn5$_{Ts_i - ref_k}$**.** There is *no significant difference* in the number of Ts_i instances before and after the application of ref_k.

Finally, as for $\mathbf{RQ}_4$, the null hypothesis was:

Hn6$_{Em_i - ref_k}$**.** There is *no significant difference* in terms of Em_i before and after the application of ref_k.

If one of the null hypotheses is statistically rejected, we will accept the corresponding alternative hypotheses, which are implicitly understood to be the opposite of the null hypotheses previously described. Then, we verified the working hypotheses by building statistical models.

Statistical modeling for $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. To address our first two research questions, we devised a *Logistic Regression Model* for each refactoring operation considered in the study. Such a model belongs to the class of Generalized Linear Models (GLM) [39] and relates a (dichotomous) dependent variable—in our case, whether or not a particular type of refactoring is performed—with either continuous and discrete independent variables—the quality and effectiveness metrics considered in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$.

Before building the statistical model, we assessed the presence of multicollinearity [41], which arises when two or more independent variables are highly correlated and can be predicted one from the other. We used the `vif` (Variance Inflation Factors) function and discard highly correlated variables, putting a threshold value equal to 5 [41].

For each statistical model, we assess whether each independent variable is significantly correlated with the dependent variable (using a significance level of $\alpha = 5\%$. Then, we quantify this correlation using the Odds Ratio (OR) [8], which is a measure of the strength of the association between each independent variable and the dependent variable. Higher OR values for an independent variable indicate a greater probability of explaining the dependent variable, i.e., a higher likelihood that a refactoring operation has been triggered by that variable. However, the interpretation of OR values varies based on the measurement scales of the independent variables: ratio scales are used for test code quality and effectiveness metrics, while categorical scales are used for test smells. For the metrics, the OR represents the likelihood that a test class will undergo refactoring for each one-unit increase in that variable. For test

smells, the OR reflects the likelihood that a smelly test class is to be involved in refactoring operations compared to a class that is not affected.

The statistical significance of the correlation between independent, i.e., test code quality metrics, effectiveness metrics, and test smells, and dependent variables allows us to accept or reject $\mathbf{Hn1}_{Qm_i - ref_k}$, $\mathbf{Hn2}_{Ts_i - ref_k}$, and $\mathbf{Hn3}_{Em_i - ref_k}$, while OR values measure the strengths of the correlations.

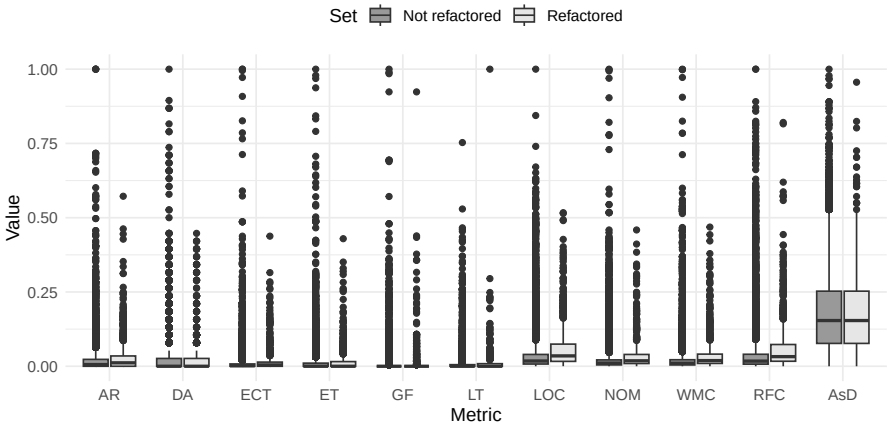
Statistical modeling for $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$. To statistically assess the impact of test refactoring operations on test code quality and effectiveness metrics and smells, we collected all the test classes subject to the refactoring type ref_k in a generic release R_i. Afterward, for each of those test classes, we computed the value of test code quality and effectiveness metrics and smells computed on the release R_i and the value of the metrics and smells computed on the release R_{i-1}.

We produced two distributions: the first representing the metric values (or the number of test smells) in R_{i-1}, i.e., before the application of ref_k; the second representing the metric values (or the number of test smells) in R_i, i.e., after the application of ref_k. On this basis, we employed the non-parametric Mann–Whitney U Test [30] (with α-value = 0.05), through which we accept or reject the null hypotheses $\mathbf{Hn4}_{Qm_i - ref_k}$, $\mathbf{Hn5}_{Ts_i - ref_k}$, and $\mathbf{Hn6}_{Em_i - ref_k}$.

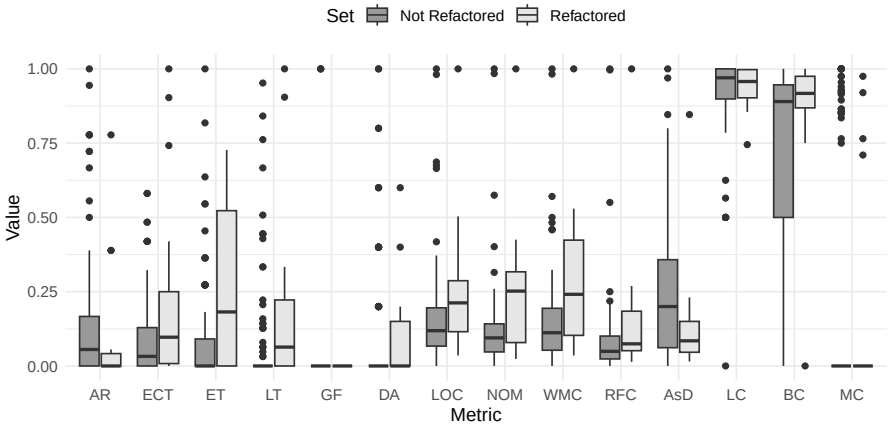
In addition, we also relied on the Vargha-Delaney ($\hat{A}_{12}$) [60] statistical test to measure the magnitude of the differences observed in the considered distributions. According to the direction and value given by $\hat{A}_{12}$, we have a practical interpretation of our findings, which depend on the test code factor considered. Specifically, if the $\hat{A}_{12}$ values are lower than 0.5, this implies that:

- The metric values computed on the release R_{i-1} are lower than those on R_i, i.e., the refactoring ref_k would have a *negative* effect on the quality metrics (Qm_i). Lower metric values in R_{i-1} would indicate that the refactoring induced the increase of these metrics in R_i, hence having a negative effect.
- The metric values computed on the release R_{i-1} are lower than those on R_i, i.e., the refactoring ref_k would have a *positive* effect on the effectiveness metrics (Em_i). Lower metric values in R_{i-1} would indicate that the refactoring induced the increase of these metric in R_i, hence having a positive effect.
- The number of test smells (Ts_i) computed on the release R_{i-1} is lower than the one computed on R_i, i.e., the refactoring ref_k would have a *negative* effect, hence suggesting that, rather than improving test code design, the refactoring induced the emergence of some form of test smells.

Similarly, a $\hat{A}_{12} > 0.50$ indicates the opposite, hence that either ref_k has a *negative* impact on the considered test code quality or effectiveness metric, or that the refactoring has a *positive* impact of the removal of test smells. Finally, $\hat{A}_{12} == 0.50$ points out that the results are identical, i.e., the refactoring has limited or no effect on the dependent variables.



(a) Dataset A with static metrics of 65 projects



(b) Dataset B with static and dynamic metrics of 12 projects

Fig. 3: Boxplots for the distributions of metrics and test smells in the datasets

5 Analysis of the Results

Figure 3 depicts the boxplots of the distributions of test smells, quality metrics, and effectiveness metrics for the sets of refactored and non-refactored tests in our datasets. Dataset A, used to investigate $\mathbf{RQ}_1$ and $\mathbf{RQ}_3$, contains 1,018 refactored test classes with 1,130 refactorings. Dataset B (for $\mathbf{RQ}_2$ and $\mathbf{RQ}_4$) has 18 refactored test classes and 20 refactorings.

5.1 Are test refactorings performed in classes with low quality? ($\mathbf{RQ}_1$)

Table 4 reports the results of the *Logistic Regression Model* for each refactoring operation ref_k analyzed in our study. As dependent variable, we considered the Qm_i quality metric in {Lines of Code (LOC), Number of Methods (NOM), Weight Method Class (WMC), Response for a Class (RFC), Assertion Density (AsD)}, the Ts_i test smell in {Assertion Roulette (AR), Duplicate Assert (DA), Handling Exception (ECT), Eager Test (ET), General Fixture (GF), Lazy Test (LT)}, and the control variables. For each variable, the table reports the value of the Estimate, the standard error (SE), and the Odds ratio (OR). We reported the Estimate and OR values in **bold** to indicate statistical significance through the $p - value < 0.05$.

When building the models, we analyzed for multi-collinearity between the independent and control variables. The control variables *releases1month*, *releases3months*, *releases6months*, and *releases12months* have a high multi-collinearity among them. It is natural for these variables to be highly correlated because releases in shorter periods will likely be included in longer periods. For example, *releases6months* is included in *releases12months*, *releases3months* is included in *releases6months* and *releases12months*, and *releases1month* is included in all of the others. Therefore, *releases1month* is less correlated to the other three variables, and it was kept in the models. A similar analysis pertains to the *commits release*. Regarding the multi-collinearity analysis for the Qm_i and Ts_i variables, the AsD metric was kept in all models along with the AR, DA, ECT, ET, and GF test smells.

With the exception of the *Replace Rule annot. w/ assertThrows* refactoring, all refactorings have at least one statistically significant variable from either the Qm_i or Ts_i set of variables. The result was quite expected since this refactoring is primarily related to the approach of handling exceptions in test code rather than directly altering the structural or complexity metrics of the code. As for the *Extract Method* refactoring, this is the only one with statistically significant variables from both sets, i.e., AsD (-0.64) metric, ET (-5.09), and GF (1.90) test smells. It means that the odds of an *Extract Method* refactoring occurring increases as the AsD metric and ET test smell values decrease, and the value of GF increases.

Concerning the Qm_i set, the AsD was statistically significant for most refactorings. In particular, it was the only significant variable for the *Add Assert Explanation* (2.74), *Extract Class* (-1.34), and *Inline Method* (-1.90) refactorings. The positive coefficient for the AsD metric could imply that when developers encounter higher values for that metric, they are more likely to opt for the *Add Assert Explanation* refactoring. The negative coefficient could imply that developers are more likely to apply *Inline Method* and *Extract Class* when the code has fewer assertions. The WMC is another Qm_i metric that is statistically significant for the *Parameterize Test* (25.52), indicating that the refactoring occurs when the methods' complexity is high.

While test smells in Ts_i are not statistically significant for most refactorings, four out of six test smells (Duplicate Assert, Eager Test, General Fixture,

Table 4: Results for the statistical model considering quality metrics (Qm_i) and test smells (Ts_i). Values in **bold** indicate the statistical significance.

Metrics	Estimate	SE	OR	Metrics	Estimate	SE	OR
Add Assert Explanation				**Parameterize Test**			
Intercept	**-6.18**	0.34	**0.00**	Intercept	**-3.21**	0.49	**0.04**
LOC release	1.25	0.84	3.48	LOC release	**-29.23**	10.66	**0.00**
contributors	-0.75	0.83	0.47	contributors	**7.21**	2.49	**1,358.45**
branches	0.26	0.44	1.29	branches	0.84	0.80	2.32
pull requests	1.09	0.67	2.96	pull requests	0.18	3.85	1.20
commits diff	-1.72	1.23	0.18	commits diff	-52.49	36.46	0.00
releases 1 month	0.52	0.69	1.67	releases 1 month	**-10.32**	3.50	**0.00**
RFC	-0.57	2.40	0.56	WMC	**25.52**	14.41	**1.2E+11**
AsD	**2.74**	0.74	**15.53**	RFC	-3.24	4.68	0.04
AR	0.55	2.63	1.73	AsD	0.24	1.33	1.27
DA	1.52	1.83	4.59	AR	2.50	16.97	12.22
ECT	1.99	2.47	7.33	DA	-26.72	23.44	0.00
GF	0.28	2.49	1.32	ECT	-1,093.04	599.96	0.00
LT	0.59	3.75	1.80	ET	-81.13	82.87	0.00
				GF	-60.55	94.08	0.00
				LT	-50.16	118.39	0.00
Extract Class				**Replace Test annot. w/ assertThrows**			
Intercept	**-4.51**	0.22	**0.01**	Intercept	**-3.65**	0.41	**0.03**
LOC release	0.98	0.60	2.66	LOC release	**-27.03**	8.51	**0.00**
contributors	**-2.04**	0.97	**0.13**	contributors	5.09	2.66	162.50
branches	**1.10**	0.27	**2.99**	branches	**1.90**	0.72	**6.71**
pull requests	0.30	0.79	1.35	pull requests	-4.08	4.65	0.02
commits diff	**1.01**	0.48	**2.75**	commits diff	-53.60	31.64	0.00
commits release	-0.28	0.66	0.75	releases 1 month	**-12.65**	3.08	**0.00**
releases 1 month	0.01	0.49	1.01	LOC	4.51	3.32	90.87
RFC	0.28	1.85	1.33	AsD	**2.17**	0.98	**8.72**
AsD	-1.34	0.68	0.26	AR	-1.53	4.33	0.22
AR	-0.60	2.42	0.55	DA	**-40.98**	17.37	**0.00**
DA	-0.21	1.73	0.81	ECT	-1.40	4.01	0.25
ECT	2.30	1.85	10.02	ET	-3.62	2.49	0.03
ET	0.91	2.17	2.49	GF	-2,543.00	1.2E+05	0.00
GF	-9.01	5.15	0.00	LT	**11.46**	5.03	**9.5E+04**
LT	2.69	2.33	14.67				
Extract Method				**Replace Rule annot. w/ assertThrows**			
Intercept	**-2.96**	0.10	**0.05**	Intercept	**-4.89**	0.46	**0.01**
LOC release	0.00	0.36	1.00	LOC release	**5.41**	0.94	**222.72**
contributors	-0.16	0.30	0.85	contributors	0.78	1.61	2.18
branches	**0.69**	0.13	**2.00**	branches	-2.46	2.00	0.09
pull requests	**-1.01**	0.41	**0.36**	pull requests	0.06	2.40	1.06
commits diff	**0.76**	0.30	**2.14**	commits diff	-24.54	18.11	0.00
commits release	0.03	0.29	1.03	releases 1 month	**-14.22**	4.05	**0.00**
releases 1 month	-0.38	0.24	0.68	AsD	0.20	1.39	1.22
RFC	**3.63**	0.59	**37.90**	AR	4.08	2.96	59.42
AsD	**-0.64**	0.31	**0.53**	DA	-2.23	3.81	0.11
AR	-0.76	0.85	0.47	ECT	-3.35	6.07	0.04
DA	1.05	0.56	2.86	ET	2.03	2.13	7.63
ECT	-0.18	0.79	0.83	GF	-5.37	9.66	0.00
ET	**-5.09**	1.46	**0.01**				
GF	**1.90**	0.65	**6.70**				
LT	2.44	1.43	11.50				
Inline Method				**Replace try/catch w/ assertThrows**			
Intercept	**-4.52**	0.24	**0.01**	Intercept	**-5.71**	0.59	**0.00**
LOC release	-0.79	0.88	0.46	LOC release	-6.98	5.51	0.00
contributors	-1.19	0.70	0.31	branches	0.74	1.16	2.09
branches	**0.81**	0.29	**2.24**	pull requests	-14.04	9.29	0.00
pull requests	-1.14	0.72	0.32	commits diff	1.48	3.46	4.41
commits diff	0.72	0.66	2.05	commits release	1.33	3.16	3.80
releases 1 month	0.69	0.47	2.00	releases 1 month	**-4.57**	2.18	**0.01**
RFC	1.53	1.54	4.64	AsD	2.21	1.42	9.07
AsD	**-1.90**	0.77	**0.15**	DA	3.58	2.77	36.01
AR	0.78	2.03	2.17	ECT	-20.31	20.81	0.00
DA	0.80	1.37	2.23	ET	**2.75**	2.75	**15.64**
ECT	0.38	2.16	1.46	GF	5.69	6.89	296.49
ET	-0.49	2.88	0.61	LT	1.83	6.07	6.24
GF	-1.80	2.54	0.17				
LT	0.60	4.01	1.82				

and Lazy Test) show significance in at least one of the models for the *Extract Method*, *Replace Test annot. w/ assertThrows*, *Replace try/catch w/ assertThrows* refactorings. It is worth noting that even though the latter two refactorings address exception handling in different ways, the ECT test smell was not found to be significant for them—which could be due to the granularity of test smells and metrics calculation. Interestingly, for the two refactorings, the DA test smell has opposite coefficients; its value is negative for the *Replace Test annot. w/ assertThrows* (-40.98) and positive for the *Replace try/catch w/ assertThrows* (3.58). Besides, the LT test smell is significant to the *Replace Test annot. w/ assertThrows* refactoring (11.46), and ET test smell for *Replace try/catch w/ assertThrows* (2.75). These findings might suggest that the former refactoring disperses concerns across different methods, and the latter consolidates them into one.

> **Answer $\mathbf{RQ_1}$.** *Except for the "Replace Rule annot. w/ assertThrows" refactoring, all other refactorings exhibit statistical significance in at least one variable from either the quality metrics (Qm_i) or test smells (Ts_i) variable sets. It underscores the complementary role of structural metrics and test smells in guiding refactoring efforts. As there is a significant difference in terms of the amount of refactoring operations (ref_k) performed on test classes having different values of Qm_i and Ts_i, we refute the $Hn1_{Qm_i - ref_k}$ and $Hn1_{Ts_i - ref_k}$.*

5.2 Are test refactorings performed in classes with low effectiveness? ($\mathbf{RQ_2}$)

In $\mathbf{RQ_2}$, we used Dataset B, i.e., the dataset in which we collected both static and dynamic metrics from 12 projects, to build a logistic model. Instead of analyzing dynamic metrics in isolation, we considered them while accounting for the overall quality of the tests considered in terms of quality metrics and test smells. The approach could help reduce bias as certain metrics coming from $\mathbf{RQ_1}$ may not be relevant anymore when accounted with dynamic metrics. At the same time, dynamic metrics may and may not be relevant depending on the quality of the tests. Therefore, despite being performed on a smaller scale, this analysis aims to provide finer-grained observations on where developers perform test refactoring.

First, we analyzed the multi-collinearity between the independent and control variables. Table 5 reports the results of the *Logistic Regression Model* for the ref_k refactoring in the set of refactoring operations considered in the study, the Em_i effectiveness metric in {Line Coverage (LC), Branch Coverage (BC), Mutation Coverage (MC)}, the Qm_i quality metric in {Lines of Code (LOC), Number of Methods (NOM), Weight Method Class (WMC), Response for a Class (RFC), Assertion Density (AsD)}, the Ts_i test smells in {Assertion Roulette (AR), Duplicate Assert (DA), Handling Exception (ECT), Eager Test (ET), General Fixture (GF), Lazy Test (LT)} and the control variables. The table shows the value of the Estimate, the standard error (SE), and the Odds

ratio (OR) for each variable. The Estimate and OR values are formatted in **bold** to indicate statistical significance through the $p - value < 0.05$.

During our analysis, we observed low data points for most refactoring operations. In particular, the *Extract Class*, *Parameterized Test*, *Replace Test annot. w/ assertThrows*, *Replace Rule with assertThrows*, and *Replace try/-catch with assertThrows* refactorings did not occur in the test classes for which the tools calculated the effectiveness metrics. In addition, the *Add Assert Argument* refactoring has few instances, and the model could not converge. Therefore, in $\mathbf{RQ}_2$, we focus on *Extract Method* and *Inline Method* refactorings.

Regarding the Em_i metrics, the BC, LC, and MC metrics compose the model of the *Extract Method* refactoring. While LC has a negative value (-1.46), BC and MC have a positive coefficient value (0.67 and 0.52). It indicates that developers find value in extracting methods even when the branch coverage is high but not when most lines have been covered by the test (i.e., low line coverage). Differently, *Inline Method* refactoring presents a negative coefficient value for MC (-43.74), indicating that the refactoring might occur when mutation coverage is low.

Concerning the other metrics, we notice that most test smells compose the logistic regression model for *Extract Method* refactoring along with two quality metrics. Besides, the NOM metric is statistically significant (4.97), indicating that refactoring occurs when there are a high number of methods. Differently, the *Inline Method* refactoring has four test smells; most of them are negative coefficient values with no statistical significance.

> **Answer $\mathbf{RQ}_2$.** *The effectiveness metrics (Em_i) did not show a statistically significant contribution to the likelihood of applying test refactorings (ref_k). Therefore, we accept $Hn3_{Em_i - ref_k}$.*

5.3 Effects of test refactorings on quality metrics and test smells ($\mathbf{RQ}_3$)

Table 6 reports the analysis of the impact of the ref_k refactoring in the set of refactoring operations considered in the study for the Qm_i quality metrics in {Lines of Code (LOC), Number of Methods (NOM), Weight Method Class (WMC), Response for a Class (RFC), Assertion Density (AsD)}. In addition, Table 7 reports the impact of the ref_k refactoring concerning the Ts_i test smells in {Assertion Roulette (AR), Duplicate Assert (DA), Handling Exception (ECT), Eager Test (ET), General Fixture (GF), Lazy Test (LT)}. For each variable, the tables report the effect size through the Vargha-Delaney statistical test ($\hat{A}_{12}$). The test indicates whether (i) there are no changes in the values of the variables before and after applying the refactoring (neutral effect, $\equiv$), (ii) the values of the variables increased after applying the refactoring (negative effect, $\downarrow$), and (iii) the values of the variables increased after applying the refactoring (positive effect, $\uparrow$). In addition, the *Mann-Whitney U test* shows whether there is a statistically significant difference between the dis-

Table 5: Results for the statistical model considering effectiveness metrics (Em_i), quality metrics (Qm_i), and test smells (Ts_i). Values in **bold** indicate the statistical significance.

Variables	Estimate	SE	OR	Variables	Estimate	SE	OR
Intercept	-5.12	2.62	0.01	Intercept	-0.11	2.74	0.90
branches	-10.30	8.03	0.00	branches	-6.03	8.23	0.00
commits diff	2.57	1.48	13.01	commits diff	-3.28	3.35	0.04
commits release	4.95	2.80	140.55	releases 6 months	-1.05	3.45	0.35
releases 1 month	4.20	2.89	66.92	MC	-43.74	27572.44	0.00
LC	-1.46	2.14	0.23	AR	-315.11	73080.58	0.00
BC	0.67	1.56	1.96	DA	-93.27	51483.14	0.00
MC	0.52	1.51	1.68	ECT	1.78	2.59	5.95
NOM	**4.97**	2.41	**143.40**	GF	-22.19	75576.33	0.00
AsD	0.17	3.31	1.18				
DA	-2.58	3.31	0.08				
ECT	1.44	1.89	4.23				
ET	2.94	3.10	19.00				
GF	-11.36	2280.79	0.00				

Table 6: Results for the effect size on the quality metrics (Qm_i).

Quality Metrics	U test	p-value	$\hat{A}_{12}$	Quality Metrics	U test	p-value	$\hat{A}_{12}$
Add Assert Explanation				**Parameterize Test**			
LOC	-0.67	0.51	0.50 ≡	LOC	1.47	0.15	0.52 ↑
NOM	0.29	0.77	0.50 ≡	NOM	1.73	0.10	0.52 ↑
WMC	-0.34	0.74	0.50 ≡	WMC	1.49	0.15	0.52 ↑
RFC	-1.39	0.17	0.50 ≡	RFC	1.67	0.11	0.52 ↑
AsD	0.00	1.00	0.50 ≡	AsD	-0.60	0.56	0.49 ↓
Extract Class				**Replace Test annot. w/ assertThrows**			
LOC	2.65	**0.01**	0.52 ↑	LOC	1.67	0.10	0.51 ↑
NOM	2.02	**0.04**	0.51 ↑	NOM	1.00	0.32	0.50 ≡
WMC	2.02	**0.05**	0.51 ↑	WMC	1.72	0.09	0.51 ↑
RFC	2.66	**0.01**	0.52 ↑	RFC	1.73	0.09	0.51 ↑
AsD	-0.48	0.63	0.50 ≡	AsD	-1.73	0.09	0.49 ↓
Extract Method				**Replace Rule annot. w/ assertThrows**			
LOC	-2.25	**0.02**	0.50 ≡	LOC	1.00	0.33	0.50 ≡
NOM	-2.84	**0.00**	0.49 ↓	NOM	NaN	NaN	0.50 ≡
WMC	-2.82	**0.01**	0.49 ↓	WMC	NaN	NaN	0.50 ≡
RFC	-2.22	**0.03**	0.50 ≡	RFC	-1.00	0.33	0.50 ≡
AsD	0.29	0.77	0.50 ≡	AsD	-1.00	0.33	0.49 ↓
Inline Method				**Replace try/catch w/ assertThrows**			
LOC	0.59	0.56	0.50 ≡	LOC	NaN	NaN	0.50 ≡
NOM	0.55	0.59	0.50 ≡	NOM	NaN	NaN	0.50 ≡
WMC	0.36	0.72	0.50 ≡	WMC	NaN	NaN	0.50 ≡
RFC	0.59	0.56	0.50 ≡	RFC	NaN	NaN	0.50 ≡
AsD	-0.42	0.68	0.50 ≡	AsD	NaN	NaN	0.50 ≡

tributions corresponding to the test code quality before and after refactorings, with the $p - value$ indicating the statistical significance.

When analyzing the values of the Qm_i quality metrics in Table 6, we notice that the values remained the same before and after applying most of the test refactorings. Although a few test classes were refactored in two consecutive releases, the values of the metrics changed after applying only two refactorings from Fowler's catalog. More specifically, the *Extract Class* refactoring improved the test classes concerning the Lines of Code (LOC), Number of Methods (NOM), Weight Method Class (WMC), and Response for a Class (RFC) metrics (0.51 and 0.52). Differently, after applying the *Extract Method* refactoring, the NOM and WMC metrics increased their values (0.49). These results were quite expected.

On the one hand, the *Extract Class* reduces the complexity and size of the original test class by extracting part of it into a new class. This can be illustrated by the extraction of a superclass and two other classes from the `BinderTest`[7] test class of the `vaadin/framework` project. Listing 1 shows the code extracted from `BinderTest`, and the generated classes. The diff highlights in red the lines removed, and in green the lines added. Originally, `BinderTest` had the metrics values: LOC = 1,311, NOM = 72, WMC = 81, and RFC = 88. All these metrics values significantly exceed the threshold values typically observed in high-quality systems by previous literature (see Table 1, i.e., LOC $\leq$ 222, NOM $\leq$ 16, WMC $\leq$ 32, and RFC $\leq$ 49; hence suggesting that the class was of low-quality. After the refactoring, instead, `BinderTest` presented significantly reduced metrics values, fitting the threshold values: LOC = 177, NOM = 18, WMC = 18, and RFC = 18.

On the other hand, *Extract Method* refactoring improves cohesion and reduces code duplication, but it might increase the complexity of the class and the number of methods. In the `NodeBasedNodeContractorTest`[8] test class of the `graphhopper/graphhopper` project, we can observe three *Extract Method* refactorings. Listing 2 shows the `createIgnoreNodeFilter` method being extracted from `testShortestPathSkipNode`, `testShortestPathSkipNode2`, and `testShortestPathLimit` methods. Before the refactoring, the test class had the metrics values: RFC = 31, WMC = 28, NOM = 22, and LOC = 395. The extraction of the method increased the quality metrics by one unit. Although these values, except for LOC, fall within the thresholds described in Table 1, the higher values of the metrics after the refactoring suggest a potential increase in complexity.

As for the test-specific refactorings, the Assertion Density metric presents a negative effect after applying the *Parameterized Test*, *Replace Test annot. w/ assertThrows* and *Replace Rule annot. w/ assertThrows* (0.49). Using test annotations does not require the method to have assertions to verify the results, but after the refactoring, new assertions are introduced. Listing 3 shows the re-

[7] Commit: `https://github.com/vaadin/framework/commit/41516b54350bdfb597d6f60961266d3c2c57b880`

[8] Commit: `https://github.com/graphhopper/graphhopper/commit/d3381bebedae15c360e1b5cadc0a9e4644de5950`

```
27      -  public class BinderTest {
   ...     // Code supressed for redability
46      -    TextField nameField;
47      -    TextField ageField;
48      -
49      -    Person p = new Person();
        13 +  public class BinderTest extends BinderTestBase<Binder<Person>, Person> {
57      14
58      15     @Before
59      16     public void setUp() {
60      17       binder = new Binder<>();
61      -        p.setFirstName("Johannes");
62      -        p.setAge(32);
63      -        nameField = new TextField();
64      -        ageField = new TextField();
        18 +     item = new Person();
        19 +     item.setFirstName("Johannes");
        20 +     item.setAge(32);
65      21     }
   ...     // Code supressed for redability
186     -    public void validate_notBound_noErrors() throws ValidationException {
   ...     // Code supressed for redability
1310    176    }
1311    177  }
```

```
        30 +  public abstract class BinderTestBase<BINDER extends Binder<ITEM>, ITEM> {
   ...     // Code supressed for redability
        42 +   protected TextField nameField;
        43 +   protected TextField ageField;
   ...     // Code supressed for redability
        52 +   @Before
        53 +   public void setUpBase() {
        54 +     nameField = new TextField();
        55 +     ageField = new TextField();
        56 +   }
        57 + }
```

```
        30 +  public class BinderValidationStatusTest extends
        31 +      BinderTestBase<Binder<Person>, Person> {
   ...     // Code supressed for redability
        36 +   @Before
        37 +   public void setUp() {
        38 +     binder = new Binder<>();
        39 +     item = new Person();
        40 +     item.setFirstName("Johannes");
        41 +     item.setAge(32);
        42 +   }
   ...     // Code supressed for redability
        507  }
```

```
        41 +  public class BinderConverterValidatorTest  extends
        42 +      BinderTestBase<Binder<Person>, Person> {
   ...     // Code supressed for redability
        687  }
```

Listing 1: Refactoring in the `BinderTest` to extract: `BinderTestBase`, `BinderValidationStatusTest`, and `BinderConverterValidatorTest`.

```
42      -  public class NodeContractorTest {
        39 +  public class NodeBasedNodeContractorTest {
        ...                                   // Code supressed for redability
80      78    public void testShortestPathSkipNode() {
81      79      createExampleGraph();
        ...                                   // Code supressed for redability
88      -       algo.setEdgeFilter(new NodeContractor.IgnoreNodeFilter(lg, graph.getNodes()).setAvoidNode(3));
        86 +    algo.setEdgeFilter(createIgnoreNodeFilter(3));
242     276
100     98    public void testShortestPathSkipNode2()
244     278     createExampleGraph();
        ...                                   // Code supressed for redability
108     -       algo.setEdgeFilter(new NodeContractor.IgnoreNodeFilter(lg, graph.getNodes()).setAvoidNode(3));
        106 +   algo.setEdgeFilter(createIgnoreNodeFilter(3));
        ...                                   // Code supressed for redability
118     116   public void testShortestPathLimit() {
        ...                                   // Code supressed for redability
124     -       algo.setEdgeFilter(new NodeContractor.IgnoreNodeFilter(lg, graph.getNodes()).setAvoidNode(3));
        122 +   algo.setEdgeFilter(createIgnoreNodeFilter(3));
        ...                                   // Code supressed for redability
        338   private IgnoreNodeFilter createIgnoreNodeFilter(int node) {
        339     return new IgnoreNodeFilter(lg, graph.getNodes()).setAvoidNode(node);
        340   }
        ...                                   // Code supressed for redability
395     397   }
```

Listing 2: Method extraction from `testShortestPathSkipNode`, `testShortestPathSkipNode2`, and `testShortestPathLimit`, generating the `createIgnoreNodeFilter` method.

placement of the annotation in the `shouldThrowAnExceptionForUnknownCode` method by an `assertThrows` (from lines 9, 13 to 10, 14). The refactoring occurs in the `ChartRangeTest`[9] class of the `iextrading4j` project. Although there are no major changes in the quality metrics before and after refactoring, the code is improved by adding an assertion method. Differently, the *Parameterized Test* allows the removal of loop structures, improving LOC, NOM, WMC, RFC metrics (0.52). The *Replace Test annot. w/ assertThrows* also allows the removal of redundant code for exception handling within test methods, improving the values of the LOC, and WMC metrics (0.51). As another example, Listing 4 shows the replacement of the try/catch block by an assertThrows. The refactoring occurs in the `EmissaryTest`[10] class of the `NationalSecurityAgency/emissary` project. Besides the decrease of the LOC from 328 to 233, almost fitting the threshold value of 222 (Table 1), we observe that WMC also decreased by 1 unit.

Focusing on Table 7, we observe that most test classes where developers applied refactorings retain the same values of test smells. As previously

[9] Commit: `https://github.com/WojciechZankowski/iextrading4j/commit/70eaf30a634b22320dc7aafb71d35fae0b7c02a3`

[10] Commit: `https://github.com/NationalSecurityAgency/emissary/commit/13588180f888b86eea71167c7ec441ecc1c25d9a`

```
 7   8      public class ChartRangeTest {
 8   9
 9       -    @Test(expected = IllegalArgumentException.class)
     10  +    @Test
10   11       public void shouldThrowAnExceptionForUnknownCode() {
11   12         final String code = "12m";
12   13
13       -       ChartRange.getValueFromCode(code);
     14  +       assertThrows(IllegalArgumentException.class, () -> ChartRange.getValueFromCode(code));
14   15       }
     ...                                    // Code supressed for redability
26   27     }
```

Listing 3: Replacement of test annotation in the `shouldThrowAnExceptionForUnknownCode` method by an `assertThrows`.

```
31   31     public class EmissaryTest extends UnitTest
32   32
33   33       @Test
34   34       public void testDefaultCommandsUnmodifiable() {
35       -      try {
36       -        Emissary.EMISSARY_COMMANDS.put("junk", new JunkCommand());
37       -        fail("Should have thrown");
38       -      } catch (UnsupportedOperationException e) {
39       -        // this is the right path
40       -      }
     35  +      assertThrows(UnsupportedOperationException.class, () -> Emissary.EMISSARY_COMMANDS
                     .put("junk", new JunkCommand()));
41   36       }
```

Listing 4: Replacement of try/catch block in the `testDefaultCommandsUnmodifiable` method by an `assertThrows`.

stated, only a few test classes were refactored in two consecutive releases. We can also observe that the *Add Assert Explanation*, *Extract Class*, and *Replace Test cannot. w/ assertThrows* refactorings led to an increase in the number of test smells. For example, after applying the *Extract Class*, the test methods that interact with the newly extracted class may become more complex and tightly coupled to the implementation details of the extracted class. Listing 1 presents the code extracted from `BinderTest` of the `vaadin/framework` project. The extraction of class attributes (lines 46 - 49) and setup (lines 61 - 64) generated the `BinderTestBase` superclass. The extraction of methods (lines 186 - 1310) generated the `BinderConverterValidatorTest`, and the `BinderValidationStatusTest` classes. Before the refactoring, the `TextField nameField` and `TextField ageField` fields were being set up together, even though individual tests may not need all of them, i.e., indicating a *General Fixture* test smell. After refactoring, the `BinderTestBase` handles the generic setup for fields, while `BinderTest` focuses on binder-specific setup.

Table 7: Results for the effect size on the test smells (Ts_i) after a refactoring.

Test Smells	U test	p-value	$\hat{A}_{12}$	Test Smells	U test	–value	$\hat{A}_{12}$
Add Assert Explanation				**Parameterize Test**			
AR	0.00	1.00	0.50 ≡	AR	-2.61	**0.02**	0.38 ↓
DA	0.81	0.42	0.50 ≡	DA	NaN	NaN	0.50 ≡
ECT	-0.62	0.54	0.49 ↓	ECT	-1.74	0.09	0.42 ↓
ET	-0.63	0.53	0.50 ≡	ET	-1.00	0.33	0.48 ↓
GF	-0.42	0.67	0.49 ↓	GF	-2.03	0.05	0.42 ↓
LT	-0.15	0.88	0.50 ≡	LT	-1.43	0.16	0.46 ↓
Extract Class				**Replace Test annot. w/ assertThrows**			
AR	-1.68	0.09	0.49 ↓	AR	-1.29	0.21	0.47 ↓
DA	-1.42	0.16	0.50 ≡	DA	-1.00	0.32	0.48 ↓
ECT	-1.38	0.17	0.49 ↓	ECT	-1.47	0.15	0.45 ↓
ET	-1.27	0.21	0.50 ≡	ET	-1.43	0.16	0.46 ↓
GF	-0.89	0.37	0.50 ≡	GF	-1.41	0.17	0.44 ↓
LT	-1.41	0.16	0.49 ↓	LT	-1.62	0.11	0.44 ↓
Extract Method				**Replace Rule annot. w/ assertThrows**			
AR	1.45	0.15	0.50 ≡	AR	1.00	0.33	0.51 ↑
DA	1.00	0.32	0.50 ≡	DA	NaN	NaN	0.50 ≡
ECT	2.30	**0.02**	0.50 ≡	ECT	NaN	NaN	0.50 ≡
ET	-0.30	0.77	0.50 ≡	ET	NaN	NaN	0.50 ≡
GF	-0.34	0.73	0.50 ≡	GF	NaN	NaN	0.50 ≡
LT	-0.09	0.93	0.50 ≡	LT	NaN	NaN	0.50 ≡
Inline Method				**Replace try/catch w/ assertThrows**			
AR	1.72	0.09	0.50 ≡	AR	NaN	NaN	0.50 ≡
DA	1.27	0.21	0.50 ≡	DA	NaN	NaN	0.50 ≡
ECT	2.12	**0.04**	0.50 ≡	ECT	NaN	NaN	0.50 ≡
ET	1.00	0.32	0.50 ≡	ET	NaN	NaN	0.50 ≡
GF	1.37	0.17	0.50 ≡	GF	NaN	NaN	0.50 ≡
LT	1.00	0.32	0.50 ≡	LT	NaN	NaN	0.50 ≡

Additionally, it is worth noticing that the *Mann-Whitney U test* shows no statistically significant difference in the numbers of test smells before and after test-specific refactorings. Inclusively, the statistical test did not perform for some refactoring types (e.g., *Parameterize Test*, *Replace Rule annot. w/ assertThrows*, and *Replace try/catch with assertThrows* refactorings), indicated by NaN values. It may occur due to the low amount of data referring to refactorings and test smells.

› **Answer RQ$_3$.** *The values of the quality metrics (Qm_i) changed after applying refactorings (ref_k) from Fowler's catalog. In particular, the Extract Class improved the coupling, cohesion, and size of the test code. Differently, after applying some test-specific refactorings, the number of test smells (Ts_i) increased. As there is a significant difference in terms of the amount of Qm_i and Ts_i, we can refute $Hn4_{Qm_i - ref_k}$ and $Hn5_{Ts_i - ref_k}$.*

Table 8: Results for the effect size on the effectiveness metrics (Em_i), quality metrics (Qm_i), and test smells (Ts_i) before and after a refactoring.

Test Smells	U test	p-value	$\hat{A}_{12}$	Test Smells	U test	p-value	$\hat{A}_{12}$
Extract Method				**Inline Method**			
LOC	-1.80	0.10	0.47 ↓	LOC	NaN	NaN	0.50 ≡
NOM	-1.84	0.09	0.47 ↓	NOM	NaN	NaN	0.50 ≡
WMC	-1.79	0.10	0.47 ↓	WMC	NaN	NaN	0.50 ≡
RFC	-1.68	0.12	0.48 ↓	RFC	NaN	NaN	0.50 ≡
AsD	-1.00	0.34	0.48 ↓	AsD	NaN	NaN	0.50 ≡
AR	NaN	NaN	0.50 ≡	AR	NaN	NaN	0.50 ≡
DA	-1.00	0.34	0.47 ↓	DA	NaN	NaN	0.50 ≡
ECT	-1.48	0.17	0.46 ↓	ECT	NaN	NaN	0.50 ≡
ET	-1.44	0.17	0.46 ↓	ET	NaN	NaN	0.50 ≡
GF	NaN	NaN	0.50 ≡	GF	NaN	NaN	0.50 ≡
LT	-1.75	0.11	0.48 ↓	LT	NaN	NaN	0.50 ≡
LC	-1.19	0.26	0.46 ↑	LC	NaN	NaN	0.50 ≡
BC	-0.68	0.51	0.49 ↑	BC	NaN	NaN	0.50 ≡
MC	NaN	NaN	0.50 ≡	MC	NaN	NaN	0.50 ≡

5.4 Effects of test refactorings on code effectiveness ($\mathbf{RQ}_4$)

In $\mathbf{RQ}_4$, we considered the smaller amount of data contained in dataset B to analyze the effects of test refactorings on the effectiveness metrics. In addition, we selected only the instances where the refactorings actually occurred. For that reason, we follow the analysis with two out of four test refactorings (*Extract Method* and *Inline Method*).

In this analysis, we assume that an increase in the Em_i effectiveness metrics in {Line Coverage (LC), Branch Coverage (BC), Mutation Coverage (MC)} would imply an improvement in the test code quality. The Vargha-Delaney statistical test ($\hat{A}_{12}$) indicates whether (i) there are no changes in the values of the variables before and after applying the refactoring (neutral effect, ≡), (ii) the values of the variables increased after applying the refactoring (positive effect, ↑), and (iii) the values of the variables decreased after applying the refactoring (negative effect, ↓). On the contrary, we assume that increasing the values of Qm_i quality metrics in {Lines of Code (LOC), Number of Methods (NOM), Weight Method Class (WMC), Response for a Class (RFC), Assertion Density (AsD)}, or Ts_i test smells in {Assertion Roulette (AR), Duplicate Assert (DA), Handling Exception (ECT), Eager Test (ET), General Fixture (GF), Lazy Test (LT)}, would reduce the test code quality. In addition, the *Mann-Whitney U test* shows whether there is a statistically significant difference between the distributions corresponding to the test code quality before and after refactorings, with the $p-value$ indicating the statistical significance.

Table 8 reports the results of the analysis of the effect size for two ref_k refactorings considered in the study. Although not performed systematically, we could observe the benefits of refactorings in the code coverage. More specifically, the *Extract Method* refactoring positively affects line and branch cov-

```
  5   11      public class JTSOpCmdTest extends TestCase {
      ...                                          // Code supressed for redability
       97 +     public void testStdInWKT() {
       98 +       runCmd( args("-a", "stdin", "-f", "wkt", "envelope"),
       99 +           stdin("LINESTRING ( 1 1, 2 2)"),
      100 +           "POLYGON" );
      101 +     }
      ...                                          // Code supressed for redability
 95  132      public void runCmd(String[] args, String expected)
 96  133      {
      134 +     runCmd(args, null, expected);
      135      }
 97  136
      137 +    private void runCmd(String[] args, InputStream stdin, String expected) {
      ...                                          // Code supressed for redability
      151 +    }
110  152    }
```

Listing 5: Extraction of `runCmd(String[], InputStream, String)` and its reuse by new methods.

erage. For example, the `runCmd(String[], InputStream, String)` method was extracted from `runCmd(String[], String)` method in the `JTSOpCmdTest` class[11] of `locationtech/jts` project to allow reusability into newly added test methods. Listing 5 shows the extraction of `runCmd(String[], InputStream, String)` (lines 137 - 151) and a new method `testStdInWKT` calling it (lines 97 - 101). The *Mann-Whitney U test* shows there is no statistically significant difference between the distributions before and after refactorings. The lack of statistical significance could be due to a limited number of instances of the test refactorings in the dataset.

> **Answer RQ$_4$.** *For the few instances with test refactorings (ref_k), we could notice an improvement in the effectiveness metrics (Em_i) for the Extract Method refactoring. However, there is no statistically significant difference in the test code quality before and after the refactorings, leading us to accept $Hn6_{Em_i - ref_k}$.*

6 Discussion and Implications

The results of the study provided us with a number of observations, reflections, and implications for research and practice. In this section, we elaborate on the main insights coming from our study.

[11] Commit: `https://github.com/locationtech/jts/commit/e989f8dba024a61387407b775ec81c9b93854db9`

6.1 Summary of the main findings

When analyzing whether the low-quality test code drives test refactorings, we observed the Assertion Density (AsD) metric is related to most test refactorings. It indicates that the more assertions in the test code, the more likely developers would refactor it. At the same time, test refactorings such as the *Parameterize Test*, *Replace Test annot. with assertThrows*, *Replace Rule with assertThrows*, and *Replace try/catch w/ assertThrows* refactorings were not related to the number of assertions because they refer to changes in the test structure or address some other non-density related concern (e.g., migration of testing framework). For example, developers usually place assertions within loop structures in the test code to assert the same condition with different values. With the *Parameterize Test* refactoring, developers remove the loop structure, but the test method continues under tests with a set of values passed as parameters. As for the negative relation of the *Extract Class* and *Extract Method* refactorings to the AsD metric, we can understand it allows classes and methods to focus on specific aspects of the test logic, leading to a more modular structure. For example, some test methods implement hard logic to stimulate the production class but have few assertions, indicating developers could place logic into another class or method to allow reusability. In conclusion of this first observation, we may argue that our findings complement previous knowledge on the properties of AsD [9,11,25]: not only a high AsD typically reduces fault proneness of production code, but it is also able to drive test refactoring in most cases.

> **Finding$_1$.** *Assertion Density is a good indicator of test refactorings. Test refactorings not driven by Assertion Density refer to changes in the test structure possibly indicating migration of testing frameworks.*

In addition, we found some unexpected results when analyzing whether the presence of test smells drives the test refactorings. The literature points out that the *Add Assert Argument* refactoring is applied to solve the Assertion Roulette (AR) test smell, and the *Replace try/catch with assertThrows* refactoring is often used to remove the Handling Exception (ECT) test smell [13]. However, no significant correlation exists between them, indicating that those refactorings can stem from variations in coding styles or project-specific guidelines. It is also interesting to notice that the *Extract Method* refactoring is negatively related to or does not have a relationship with the test smells responsible for indicating the method has spread or tangled concerns, i.e., the Handling Exception (ECT) and Lazy Test (LT) test smells. On the one hand, these findings seem to call for further research on the motivations behind the application of refactoring operations: this is indeed a current knowledge gap, as previous work solely focused on the reasons driving production code refactoring [50]. On the other hand, our observations may potentially suggest improvements for test refactoring recommenders, e.g., information coming from the projects' guidelines may empower the support provided to practitioners.

Finding$_2$. *Test smell-driven refactorings lack consistent correlation, suggesting diverse motivations beyond test smell removal.*

Finally, the analysis of whether the test refactorings improve the code quality shows that most test classes maintained stable metric values. Still, the *Extract Class*, *Parameterized Test*, and *Replace Test annot. w/ assertThrows* refactorings improve most quality metrics. Another interesting finding concerns the *Extract Method* refactoring. While it might not be driven by improving quality metrics or test smells, it shows an improvement in the effectiveness metrics. It highlights the trade-offs when applying this refactoring operation. These observations highlight a key, additional property of test refactoring: our findings indeed reveal that even if not performed systematically, test refactoring may be beneficial for the effectiveness of test code. In this respect, our results may encourage further research on multi-objective test refactoring recommendations and prioritization. These approaches aim not only to enhance test code quality but also to address the effects on code and mutation coverage. Additionally, our results may raise practitioners' awareness of the advantages of test refactoring, making them more inclined to adopt systematic strategies for conducting regular refactoring initiatives.

Finding$_3$. *Test refactoring has a positive effect on effectiveness and, even if not performed systematically, we could already observe the benefits of extracting methods for line and branch coverage.*

6.2 Implications and Lessons Learned

Interplay of Effectiveness and Quality Metrics. The *Extract Method* refactoring is a compelling example of this dynamic. Our analysis shows that quality metrics and test smells drive such refactoring. In addition, the refactoring positively impacts effectiveness metrics and negatively impacts quality metrics. As a practical implication, developers can strategically choose code sections for extraction to enhance code cohesion and reduce test smells. More specifically, developers could prioritize classes with lower Assertion Density (AsD) metric and a higher presence of the General Fixture (GF) test smell for extraction. This highlights the importance of a balanced approach to refactoring, where developers should carefully weigh the benefits of making the decision to refactor the test code. These findings serve as valuable inputs for researchers in the field of refactoring and test code quality. The trade-offs between quality and effectiveness have not yet been extensively explored in the context of test refactoring. This gap represents a promising research opportunity, with the potential to develop more practical and pragmatic test refactoring recommenders and prioritizers that could selectively suggest refactoring practices to optimize both quality and effectiveness.

👉 **Lesson$_1$.** *Refactoring may not always have a positive impact on code quality and effectiveness. Refactoring recommenders could help developers analyze the trade-offs of applying a refactoring operation.*

Impact of Test Refactorings on Code Quality. While most test classes maintain stable metric values post-refactoring, *Extract Class* and *Replace Test annot. w/ assertThrows* refactorings improve quality metrics. Differently, *Extract Method* refactoring improves the effectiveness metrics and reduces quality metrics, highlighting trade-offs associated with this refactoring operation. Therefore, understanding the impact of different refactorings on code quality metrics informs developers about potential trade-offs and benefits associated with specific refactorings. This knowledge can guide decision-making processes during the refactoring phase, allowing developers to choose which refactoring operation leads to tangible improvements in code quality and effectiveness. Our findings, therefore, open the way to further empirical investigations into the properties of specific test code refactoring operations: we deem these empirical studies instrumental to the definition of improved refactoring recommendation and prioritization strategies.

👉 **Lesson$_2$.** *Deciding which refactoring operation is best in a context is not easy. More empirical research on the peculiarities of each test code refactoring practice may shed light on the strategies to adopt when implementing refactoring recommendation and prioritization approaches.*

Unexpected Relationship Between Test Smells and Refactorings. There is no significant correlation between certain test smells and corresponding refactorings, such as *Assertion Roulette* with *Add Assert Argument*, and *Handling Exception* with *Replace try/catch with assertThrows*. These unexpected relationships suggest that refactorings may not always directly address or resolve associated test smells. In relation to this lesson learned, we may find multiple actionable implications. In the first place, practitioners and researchers may consider project-specific contexts and coding styles when interpreting the relationships between test smells and refactorings. This may not only optimize the current quality assurance practices applied by developers, but also suggest improvements in the way refactoring recommendations should be provided. We may envision further studies on the role played by contextual, project-specific indicators on the quality of refactoring recommendations provided to practitioners. In the second place, our results clearly indicate the need for more empirical investigations into the actual motivations driving test code refactoring decisions. Our study suggests the existence of alternative motivations that go beyond test code quality and effectiveness. An improved understanding of these alternative motivations may enlarge the knowledge of the practices applied by developers, possibly providing further insights on how to design effective refactoring recommenders and prioritizers.

👉 **Lesson$_3$.** *Contextual factors might influence the test smell-refactoring relationships. Therefore, developers and researchers should consider project specifics and coding styles when refactoring test code.*

Challenges of Mining Static and Dynamic Metrics. As a final point of discussion, let us elaborate on the challenges faced when applying mining software repository techniques to the analysis of test code change history. In particular, our tooling required the compilation of past snapshots of the systems in order to execute test suites and compute test code effectiveness indicators. Unfortunately, we failed in most cases, implying a notable reduction of the sample and, consequently, of the generalizability of our findings. The challenges of analyzing change history information have been already explored by researchers in the past [29,58]: we could corroborate them, showing that the mining of dynamic metrics failed in ≈93% of the projects, i.e., we could analyze only 12 projects out of the 175.

Furthermore, integrating data from several automated tools for measuring code quality, detecting test smells, and mining refactoring operations presents challenges in achieving uniform metric computation and dataset consistency. Dependency issues and tool-specific requirements can also hinder metrics calculation. While we addressed these issues by establishing traceability links between the tools and creating separate datasets based on available metrics, some valuable data may have been overlooked. Additionally, automated tools should be robust enough to handle diverse project environments and dependencies for comprehensive metric computation.

👉 **Lesson$_4$.** *Mining dynamic information from change history is challenging and may threaten the consistency of the dataset collected. Automated tools that mix quality and effectiveness metrics are needed to allow a more comprehensive analysis of the test code quality.*

7 Threats to validity

This section discusses the potential threats that may affect the validity of our empirical study plan.

Construct validity. When considering the relationship between theory and observation, a first threat concerns the criteria we used to select software projects: despite the actions to standardize the building process, we had configuration problems between PITEST, JACOCO and SUREFIRE. Following the recent insight on how to fix build failures [29,58], we set the `<argLine>` property in the header of the MAVEN configuration file fixing the problem.

Concerning the independent variables used to assess the test code quality, we did not compute all the Chidamber & Kemerer metrics because some of them cannot be applied to the context of test code (e.g., *Depth Inheritance Tree*). Nevertheless, we have chosen a mix of metrics capturing the test code size, structural, and dynamic characteristics. A possible threat to validity con-

cerns the identification of the independent and dependent variables through automated tools. We are aware of the possible noise that might be introduced in terms of false positives. To partially mitigate this threat, we selected tools already validated and used by the research community [31, 43, 45].

When computing test smells, we linked test classes to their corresponding production classes using a pattern-matching approach based on naming conventions and class hierarchy [45]. While more advanced methods, such as dynamic slicing, offer higher accuracy, our approach strikes a balance between accuracy and scalability. However, we did take precautions. Specifically, the pattern-matching approach can produce false positive links if multiple production classes share the same name but exist in different paths. In our study, this issue did not arise as there were no production classes with identical names but different paths. However, future replications of our study on different systems may need to address this potential issue to enhance the linking capabilities of the pattern-matching approach.

Finally, a potential threat to the validity of our findings stems from integration challenges between PITEST and SUREFIRE. Specifically, some projects that build successfully without PITEST encounter failures during the mutation testing stage. This issue arises from the lack of direct integration between the two tools, leading to several common problems: (i) tests excluded by SUREFIRE may still be executed by PITEST, potentially skewing mutation testing results; (ii) tests that rely on environment variables or configurations set by SUREFIRE but not replicated in PITEST may fail, causing inconsistencies; and (iii) hidden dependencies on the order of test execution may surface if PITEST runs the tests in a different sequence than SUREFIRE. These discrepancies underscore the importance of careful configuration when using PITEST with SUREFIRE to ensure reliable and consistent mutation testing outcomes.

Internal Validity. This category of threats to validity concerns by-product changes of other maintenance activities (e.g., bug fixes or changes in requirements) that could also contribute to the removal of test smells. Therefore, the data analysis might not indicate a causal relationship; rather, there is a possibility of a relationship that may be further investigated. We corroborated our quantitative results through some qualitative insights. In addition, we acknowledge test flakiness as a potential threat to the internal validity, which can impact the reliability of our findings. However, despite being a severe issue for practitioners, previous investigations found test flakiness to arise in a limited amount of cases, e.g., Luo et al. [28] found out that flaky tests affect up to 4.56% of test cases. In this sense, it is reasonable to believe that the problem of test flakiness has a limited impact on our findings.

External Validity. This class of threats to validity mainly concerns the subject projects of our study. We selected open-source projects developed with JAVA 8, which are only a fraction of the complete picture of open-source software and do not necessarily represent industrial practices. Therefore, the results may not be generalized to the industrial context or other programming languages. In addition, we have selected projects based on the number of stars,

which may raise some popularity bias. We partially mitigated this threat by selecting open-source projects coming from different contexts and organizations. However, replications of our work would be beneficial to corroborate our findings in different contexts: to stimulate further research, we release all materials and scripts as part of an archived online appendix [33].

Finally, we recognize that limit our research to Java 8 may represent a potential threat to validity. Although this is still the most popular version among developers, newer versions have introduced language features and performance optimizations that may affect how refactoring operations and test code quality are approached. This could influence the applicability of our results to projects using later versions of Java. Replicating our study with more recent versions would be beneficial in verifying whether similar correlations exist.

Conclusion validity. To address how frequently test refactoring is performed on test classes affected by quality or effectiveness concerns, we have used logistic regression models to identify correlations. Other than highlighting significant correlations, we have reported and discussed OR values. In addition, to investigate the effect of test refactoring on test code quality and effectiveness, we have employed well-established statistical tests such as the Mann–Whitney Test [30] and the Vargha-Delaney Test ($\hat{A}_{12}$) [60]. Our analysis was conducted at the granularity of classes because the tools used to extract variables work at this level. This may bias our conclusions, as this granularity may be subject to various confounding variables. On the one hand, this is a limitation that we, unfortunately, share with all the other research works that analyze dynamic test code metrics [26]. On the other hand, we have included multiple process- and project-level control variables, through which we have partially mitigated this threat to validity.

An additional point to remark is that our data collection procedure cannot distinguish between changes that were meant as refactoring and other changes where refactoring was applied as part of other modifications. We might have mitigated this limitation by extracting refactoring changes through the analysis of issues and pull requests, i.e., collecting changes explicitly intended as refactoring. Nonetheless, such an alternative method could have biased even further the conclusions drawn for two reasons connected to the availability and reliability of the information available within the developers' discussions on GitHub. More particularly:

Availability. Previous studies established that developers perform "floss refactoring", combining refactoring operations and behavioral change edits within individual commits [37]. From a practical standpoint, this means that developers do not often apply refactoring for the sake of refactoring source code, but as an instrument to perform other changes, e.g., to simplify a piece of code before making further evolutionary changes. As such, it is unlikely to find "pure" refactoring changes or discussions, in the form of issues or pull requests, around refactoring operations to be applied. To further elaborate on the matter, we computed the percentage of pure refactoring vs. non-pure refactoring commits. As a result, we found 20.1% pure refactorings in the

test code, i.e., refactorings restructuring only the test code without changing anything in the production code. This finding corroborates previous research on the matter [37], underscoring the practical challenges of isolating refactorings in real-world software development scenarios.

Reliability. Literature found that developers not only rarely document refactoring activities explicitly [61,62], but also that when they do, they are inconsistent [3], i.e., labeling changes as refactoring, although no refactoring is done at all. Other researchers discovered that the term "refactoring" is misused, i.e., developers do not often correctly distinguish between refactoring changes and normal code modifications [14]. In this respect, the seminal paper by Murphy-Hill et al. [38] reported that *"messages in version histories are unreliable indicators of refactoring activities. This is due to the fact that developers do not consistently report/document refactoring activities"*. This observation was also backed up by the findings obtained by Ratzinger et al. [47], who discovered that the extraction of refactoring documentation from repositories may lead to several false positives, as the words used by developers are too generic and do not often refer to real refactoring operations.

As a consequence, the analysis of issues and pull requests would have led to unreliable conclusions. On the contrary, the goal of a statistical study is exactly identify hidden relations between dependent and independent variables while controlling for possible confounding effects [19]: we believe that such an approach better fits our research goals. Through a large-scale, statistical investigation, we may indeed end up discovering the intrinsic factors associated with the refactoring actions performed by developers, finally providing evidence of how test refactoring is done in practice.

Finally, a potential threat to conclusion validity in this study lies in the use of static analysis tools such as SonarQube, Checkstyle, and PMD, which may influence developers' propensity to refactor test code. Given the varied implementation and usage of these tools, their influence may not be consistent across all projects and developers. While our study did not explicitly target these tools, developers might still indirectly improve test code by addressing general issues highlighted by the tools. The commitment of some developers to clean code practices, which cannot be systematically measured, could also confound the results, leading to a potential misinterpretation of the tools' impact on quality assurance.

8 Conclusion

The ultimate goal of our work was to understand whether the test code quality and effectiveness provide indications of which test classes are more likely to be refactored and to what extent test refactoring operations can improve the test code quality and effectiveness. We conducted this study on a set of 65 open-source Java projects, starting from the collection of data on the test code quality, test smells, and refactoring operations arising in the major releases of

the projects. Then, we employed statistical approaches to address the goals of our investigation and, based on the conclusions, we finally provided actionable items and implications for researchers and practitioners.

The key findings of our study reveal that test refactoring operations typically address low-quality test cases identified by test smells and quality metrics, enhancing the coupling and cohesion of test classes. However, no statistically significant correlation between test refactoring and dynamic metrics was found. Based on these findings, we identified several lessons learned and implications for researchers and practitioners.

To sum up, our paper provided the following contributions:

1. An empirical understanding of the factors triggering test refactoring operations, which comprises an analysis of how test code quality and effectiveness come into play;
2. Evidence and analysis of the impact of test refactoring on test code quality and effectiveness;
3. An online appendix [33] in which we provide all material and scripts employed to address the goals of the study.

The main considerations and conclusions of the study represent the input for our future research agenda. We plan to develop novel refactoring recommenders that explore the trade-off between test code quality and effectiveness. Furthermore, we plan to investigate the extent to which developers are willing to perform test code refactorings, including examining project-specific refactoring guidelines, such as contribution policies. In addition, we aim to explore the long-term impacts of test refactoring on project maintenance and evolution, analyzing how continuous refactoring influences the overall health and sustainability of software projects. Finally, we will also explore how to integrate dynamic metrics into refactoring tools, investigating advanced techniques that might reveal hidden patterns and benefits not apparent in our initial analysis.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001; FAPESB grants BOL0188/2020 and PIE0002/2022; CNPq grants 315840/2023-4 and 403361/2023-0; This work has also been partially funded by the European Union under NextGenerationEU with the *RECHARGE* research project, which has been funded by MUR PRIN 2022 PNRR program (Code: P2022SELA7). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or The European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript includes data as electronic supplementary material. In particular, datasets generated and analyzed during the current study, detailed results, as well as scripts and additional resources useful for reproducing the study, are available as part of our online appendix on Figshare: `https://doi.org/10.6084/m9.figshare.23666736`.

Credits

Luana Martins: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Valeria Pontillo**: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Heitor Costa**: Supervision, Resources, Writing - Review & Editing. **Filomena Ferrucci**: Supervision, Resources, Writing - Review & Editing. **Fabio Palomba**: Supervision, Resources, Writing - Review & Editing. **Ivan Machado**: Supervision, Resources, Writing - Review & Editing.

References

1. Al Dallal, J.: Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and Software Technology **58**, 231–249 (2015)
2. Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M.W., Ouni, A., Newman, C.D., Ghallab, A., Ludi, S.: Test smell detection tools: A systematic mapping study. Evaluation and Assessment in Software Engineering, p. 170–180. ACM, New York, NY, USA (2021)
3. AlOmar, E.A., Peruma, A., Mkaouer, M.W., Newman, C., Ouni, A., Kessentini, M.: How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications **167**, 114176 (2021)
4. Azeem, M.I., Palomba, F., Shi, L., Wang, Q.: Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. The Journal of Systems and Software **108**, 115–138 (2019)
5. Baqais, A.A.B., Alshayeb, M.: Automatic software refactoring: a systematic literature review. Software Quality Journal **28**(2), 459–502 (2020)
6. Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O.: When does a refactoring induce bugs? an empirical study. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, pp. 104–113. IEEE (2012)
7. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Recommending refactoring operations in large software systems. Recommendation Systems in Software Engineering pp. 387–419 (2014)

8. Bland, J.M., Altman, D.G.: The odds ratio. Bmj **320**(7247), 1468 (2000)
9. Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F.: How the experience of development teams relates to assertion density of test classes. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 223–234. IEEE (2019)
10. Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., Garcia, A.: How does refactoring affect internal quality attributes? a multi-project study. In: Proceedings of the XXXI Brazilian Symposium on Software Engineering, SBES '17, p. 74–83. ACM, New York, NY, USA (2017)
11. Chen, J., Bai, Y., Hao, D., Zhang, L., Zhang, L., Xie, B.: How do assertions impact coverage-based test-suite reduction? In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 418–423. IEEE (2017)
12. Damasceno, H., Bezerra, C., Coutinho, E., Machado, I.: Analyzing test smells refactoring from a developers perspective. In: Proceedings of the XXI Brazilian Symposium on Software Quality, SBQS '22. ACM, New York, NY, USA (2023)
13. Deursen, A., Moonen, L.M., Bergh, A., Kok, G.: Refactoring test code. Tech. rep., Centre for Mathematics and Computer Science, NLD (2001)
14. Di, Z., Li, B., Li, Z., Liang, P.: A preliminary investigation of self-admitted refactorings in open source software (s). In: Int.l Conf. on Software Engineering and Knowledge Engineering, vol. 2018, pp. 165–168. KSI Research Inc. and Knowledge Systems Institute Graduate School (2018)
15. Di Penta, M., Bavota, G., Zampetti, F.: 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, pp. 556–567 (2020)
16. Du Bois, B., Demeyer, S., Verelst, J.: Refactoring - improving coupling and cohesion of existing code. In: 11th Working Conf. on Reverse Engineering, pp. 144–151 (2004)
17. Ferreira, I., Fernandes, E., Cedrim, D., Uchôa, A., Bibiano, A.C., Garcia, A., Correia, J.a.L., Santos, F., Nunes, G., Barbosa, C., Fonseca, B., de Mello, R.: The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, p. 406–407. ACM, New York, NY, USA (2018). DOI 10.1145/3183440.3195030
18. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., USA (1999)
19. Freedman, D.A.: Statistical models: theory and practice. Cambridge University Press (2009)
20. Grano, G., De Iaco, C., Palomba, F., Gall, H.C.: Pizza versus pinsa: On the perception and measurability of unit test code quality. In: 2020 IEEE Int.l Conf. on Software Maintenance and Evolution (ICSME), pp. 336–347. IEEE (2020)
21. Guerra, E.M., Fernandes, C.T.: Refactoring test code safely. In: Int.l Conf. on Software Engineering Advances (ICSEA 2007), pp. 44–44. IEEE, New York, NY, USA (2007)
22. Iannone, E., Codabux, Z., Lenarduzzi, V., De Lucia, A., Palomba, F.: Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. Empirical Software Engineering **28**(4), 89 (2023)
23. Kim, D.J., Chen, T.H.P., Yang, J.: The secret life of test smells-an empirical study on test smell evolution and maintenance. Empirical Software Engineering **26**(5), 1–47 (2021)
24. Kochhar, P.S., Lo, D., Lawall, J., Nagappan, N.: Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Transactions on Reliability **66**(4), 1213–1228 (2017)
25. Kudrjavets, G., Nagappan, N., Ball, T.: Assessing the relationship between software assertions and faults: An empirical investigation. In: 2006 17th International Symposium on Software Reliability Engineering, pp. 204–212. IEEE (2006)
26. Kumar Chhabra, J., Gupta, V.: A survey of dynamic software metrics. Journal of computer science and technology **25**, 1016–1029 (2010)
27. Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G.: Code smells and refactoring: A tertiary systematic review of challenges and observations. The Journal of Systems and Software **167**, 110610 (2020)

28. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT Int.l Symposium on Foundations of Software Engineering, pp. 643–653 (2014)
29. Maes-Bermejo, M., Gallego, M., Gortázar, F., Robles, G., Gonzalez-Barahona, J.M.: Revisiting the building of past snapshots—a replication and reproduction study. Empirical Software Engineering **27**(3), 65 (2022)
30. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp. 50–60 (1947)
31. Martins, L., Costa, H., Ribeiro, M., Palomba, F., Machado, I.: Automating test-specific refactoring mining: A mixed-method investigation. In: Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (2023)
32. Martins, L., Pontillo, V., Costa, H., Ferrucci, F., Palomba, F., Machado, I.: Test code refactoring unveiled: Where and how does it affect test code quality and effectiveness? arXiv preprint arXiv:2308.09547 (2023)
33. Martins, L., Pontillo, V., Costa, H., Ferrucci, F., Palomba, F., Machado, I.: [Dataset] Test Code Refactoring Unveiled: Where and How Does It Affect Test Code Quality and Effectiveness? (2024). DOI https://doi.org/10.6084/m9.figshare.23666736
34. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
35. Meszaros, G., Smith, S.M., Andrea, J.: The test automation manifesto. Extreme Programming and Agile Methods - XP/Agile Universe 2003, pp. 73–81. Springer Berlin Heidelberg, Springer, Berlin, Heidelberg (2003)
36. Micco, J.: The state of continuous integration testing@ google (2017)
37. Murphy-Hill, E., Black, A.P.: Why don't people use refactoring tools? In: Proceedings of the 1st Workshop on Refactoring Tools, pp. 61–62 (2007)
38. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering **38**(1), 5–18 (2011)
39. Nelder, J.A., Wedderburn, R.W.: Generalized linear models. Journal of the Royal Statistical Society: Series A (General) **135**(3), 370–384 (1972)
40. Oliveira, P., Valente, M.T., Lima, F.P.: Extracting relative thresholds for source code metrics. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 254–263. IEEE (2014)
41. O'brien, R.M.: A caution regarding rules of thumb for variance inflation factors. Quality & quantity **41**, 673–690 (2007)
42. Papadakis, M., Shin, D., Yoo, S., Bae, D.H.: Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: Proceedings of the 40th Int.l Conf. on Software Engineering, pp. 537–548 (2018)
43. Pecorelli, F., Di Lillo, G., Palomba, F., De Lucia, A.: Vitrum: A plug-in for the visualization of test-related metrics. In: Proceedings of the Int.l Conf. on Advanced Visual Interfaces, AVI'20. ACM, New York, NY, USA (2020)
44. Pecorelli, F., Palomba, F., De Lucia, A.: The relation of test-related factors to software quality: a case study on apache systems. Empirical Software Engineering **26**, 1–42 (2021)
45. Peruma, A., Almalki, K., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F.: tsdetect: an open source test smells detection tool. Symposium on the Foundations of Software Engineering. ACM (2020)
46. Peruma, A., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F.: An exploratory study on the refactoring of unit test files in android applications. In: Proceedings of the 42nd Int.l Conf. on Software Engineering Workshops, ICSEW'20, p. 350–357. ACM, New York, NY, USA (2020)
47. Ratzinger, J., Sigmund, T., Gall, H.C.: On the relation of refactorings and software defect prediction. In: Proceedings of the 2008 Int.l working Conf. on MSR, pp. 35–38 (2008)
48. Schweikl, S., Fraser, G., Arcuri, A.: Evosuite at the sbst 2022 tool competition. In: Proceedings of the 15th Workshop on Search-Based Software Testing, pp. 33–34 (2022)
49. Shatnawi, R.: Deriving metrics thresholds using log transformation. Journal of Software: Evolution and Process **27**(2), 95–113 (2015)

50. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering, pp. 858–870 (2016)
51. Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., Fernandes, L., Garcia, A., Fonseca, B., Santos, A.: Refactoring test smells: A perspective from open-source developers. In: Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, SAST 20, p. 50–59. ACM, New York, NY, USA (2020)
52. Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., Santos, A.M.: Refactoring test smells with junit 5: Why should developers keep up-to-date. IEEE Transactions on Software Engineering pp. 1–1 (2022)
53. Sobrinho, E.V.P., De Lucia, A., de Almeida Maia, M.: A systematic literature review on bad smells–5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering **47**(1), 17–66 (2018)
54. Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A.: On the relation of test smells to software code quality. In: 2018 IEEE Int.l Conf. on Software Maintenance and Evolution (ICSME), pp. 1–12. IEEE, New York, NY, USA (2018)
55. Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A.: On the relation of test smells to software code quality. In: 2018 IEEE Int.l Conf. on Software Maintenance and Evolution (ICSME), pp. 1–12 (2018)
56. Spadini, D., Schvarcbacher, M., Oprescu, A.M., Bruntink, M., Bacchelli, A.: Investigating severity thresholds for test smells. In: Proceedings of the 17th Int.l Conf. on Mining Software Repositories, MSR '20, p. 311–321. ACM, New York, NY, USA (2020)
57. Tsantalis, N., Ketkar, A., Dig, D.: Refactoringminer 2.0. IEEE Transactions on Software Engineering **48**(3), 930–950 (2022)
58. Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process **29**(4), e1838 (2017)
59. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 403–414. IEEE (2015)
60. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)
61. Weißgerber, P., Biegel, B., Diehl, S.: Making programmers aware of refactorings. In: WRT, pp. 58–59 (2007)
62. Weißgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: 21st IEEE/ACM international conference on automated software engineering (ASE'06), pp. 231–240. IEEE (2006)
63. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
64. Wu, H., Yin, R., Gao, J., Huang, Z., Huang, H.: To what extent can code quality be improved by eliminating test smells? In: 2022 Int.l Conf. on Code Quality (ICCQ), pp. 19–26. IEEE, New York, NY, USA (2022)