Smart Prediction for Test Smell Refactorings

Luana Martins[1,3], Heitor Costa (coorientador)[2],
Fabio Palomba (coorientador)[3], Ivan Machado (orientador)[1]

[1]Federal University of Bahia (UFBA) – Brazil

[2]Federal University of Lavras (UFLA) – Brazil

[3]University of Salerno (UNISA) – Italy

{martins.luana, ivan.machado}@ufba.br, heitor@ufla.br, fpalomba@unisa.it

Abstract. *Software plays a critical role in modern society, making software testing essential for ensuring quality. Poor testing practices, known as test smells, reduce test code quality and maintainability. Although refactoring is a widely used technique to address these issues, little is known about when developers choose to refactor test smells and whether these actions actually improve test code quality. This study introduces a machine learning approach to guide test smell refactoring. First, we aim to mine refactorings performed by developers to derive a catalog of test-specific refactorings. Findings show testing framework evolution helps address test smells. Second, we aim to understand whether developers target low-quality test codes to perform refactorings and their effects on code quality. Our findings reveal that developers tend to refactor structurally low-quality test code more often. Third, we aim to learn whether developers would perform refactorings and which refactorings they would apply to improve the test code quality. Results show that Support Vector Machine models predicted refactoring decisions with 30–100% accuracy.*

1. Introduction

Software testing is a fundamental activity for software quality assurance and consists of developing helpful test cases to find bugs caused by code changes [Garousi and Küçük 2018, Bell et al. 2018]. However, software companies often do not prioritize software testing activities because they face difficulties selecting qualified personnel and allocating resources [Melo et al. 2020, Martins et al. 2021b]. Additionally, the testing activities, when carried out manually, consume much time and effort. The effort to perform testing activities depends on the artifacts under test, e.g., large and complex production classes require more effort to develop the test codes [Terragni et al. 2020].

The complexity of the software under test, the lack of expertise, and time pressure can lead software developers to use bad practices to either design or implement the test code, resulting in the so-called test smells [Deursen et al. 2001, Tufano et al. 2016, Silva Junior et al. 2020]. The presence of test smells can negatively affect the test code quality, harming the software testing and maintenance activities [Bavota et al. 2015, Palomba et al. 2016, Spadini et al. 2020, Campos et al. 2021]. Test smells differ from code smells defined by [Fowler 1999] for the production code. Specifically, the test smells reflect problems in the test cases' organization, implementation, and interaction with one another [Deursen et al. 2001]. Therefore, test smells can require test

code refactoring operations that differ from the ones applied in the production code [Deursen et al. 2001, Soares et al. 2020].

To fix test smells, developers should refactor the test code in a way that does not change the test logic [Fowler 1999]. While most refactoring recommendation tools target the production code, little attention has been devoted to detecting test smells and refactoring the test code [Aljedaani et al. 2021]. Prior test smells detection and refactoring strategies, based on rules and metrics, were simplistic and not derived from common practices used by developers [Peruma et al. 2022]. In particular, those strategies mostly rely on predefined thresholds or rules to detect test smells. However, deciding whether a refactoring operation fixes a particular test smell requires developers' expertise and intuition [Spadini et al. 2020].

For example, the *Assertion Roulette* test smell occurs when a test case contains multiple assertions without explanatory messages, making it difficult to identify the cause of a test failure [Deursen et al. 2001]. Existing rule-based tools (*e.g.*, tsDetect [Peruma et al. 2020a]) detect *Assertion Roulette* by counting assertions that lack contextual explanations. To eliminate this test smell, previous research suggests two main approaches: (i) adding explanatory messages (*e.g.*, comment blocks or assertion parameters) and (ii) refactoring the test case into a suite of single-condition tests [Deursen et al. 2001]. However, existing tools [Aljedaani et al. 2021] only assist in detection and require manual intervention for refactoring. Furthermore, they do not offer multiple refactoring alternatives for a given test smell.

Therefore, it is important for developers to have proper guidelines for identifying test smells and refactoring the test code. This aligns with this study proposal that aims to analyze supervised Machine Learning (ML) algorithms to classify developers' intentions in applying test refactoring and the specific test refactoring operation. Currently, our understanding is based on the work of [Aniche et al. 2022], which investigates the effectiveness of ML algorithms in predicting refactorings in production code. While these algorithms have been effective in improving production code, their ability to enhance test code remains uncertain. This is because test code serves a different purpose: it verifies whether software works correctly but is not part of the final product itself. First, the characteristics used to determine when and how to improve production code may not apply to test code, leading to incorrect or ineffective modifications. Second, widely accepted guidelines for improving code, such as Fowler's catalog [Fowler 1999], do not cover test-specific improvements, even though researchers like [Deursen et al. 2001] have identified them. As a result, existing tools may fail to detect or suggest necessary refinements in test code, leaving tests unclear, unreliable, or incomplete. This gap is critical in practice: if poorly structured tests miss defects in software, real-world failures can occur — ranging from minor glitches in applications to severe malfunctions in safety-critical systems such as healthcare devices or financial platforms. Ensuring high-quality test code is essential to prevent undetected errors from propagating into deployed software.

This research contributes to Computer Science by integrating Software Engineering and Machine Learning to improve test code maintenance through automated refactoring support. It introduces a *catalog of test-specific refactorings* to support developers' decision making on whether and how to refactor test code. Additionally, it investigates the *relationship between test code quality and refactoring* likelihood, demonstrating that

low-quality test code is more prone to refactoring. Finally, it applies *ML models to predict developers' refactoring decisions* to improve the overall quality of the test code. In summary, the contributions of this study might positively impact the developers' effectiveness on testing and maintenance activities (*social aspect*), the development of automated tools to assist in test refactoring (*technical aspect*), the understanding of refactoring practices and their effects on software quality (*scientific aspect*), and the efficiency of software development, potentially reducing time and resource consumption (*environmental aspect*).

2. Research Statement and Research Questions

In this work, we investigated ML techniques to classify the developers' intention to apply test refactoring and which test-specific refactoring operations they would apply. Our investigation is built upon two objectives:

1. **Mining common test smells and their refactorings in practice.** Existing test smells and test refactoring catalogs are primarily based on the researchers' intuition [Fowler 1999, Deursen et al. 2001, Meszaros 2007]. Our approach provides a catalog of test smells and test refactoring operations to fix them based on actual development practices. We explored the change history of software repositories to extract the test refactoring operations performed on the test code. Analyzing test refactoring operations helps reveal (a) the actual issues developers deemed as test smells and (b) how such issues were fixed (i.e., refactored) in practice.
2. **Identifying refactoring opportunities in test code and proper fixings for test smells.** We derived a dataset containing smelly and non-smelly refactored test codes while we mined commonly used test code refactoring in practice. Then, we used ML techniques to learn from the features extracted from the change history of the test code stored in that dataset.

To address the outlined objectives, we formulated three main Research Questions (RQ):

RQ$_1$. *How do developers perform test code refactorings to fix test smells in open-source projects?*

In **RQ**$_1$, our objective was to catalog test code refactorings commonly performed by developers in practice to address test smells. To achieve this, we conducted a qualitative study [Martins et al. 2024a] on a sample of open-source projects to manually classify test code changes. Additionally, we investigated developers' preferences regarding refactoring operations for fixing test smells by submitting pull requests to open-source projects.

RQ$_2$. *How do test refactoring operations affect test code quality?*

In **RQ**$_2$, our objective was twofold: i) to determine whether low-quality test classes, as measured by structural metrics and the presence of test smells, are more likely to have test refactorings; ii) to assess the impact of test code refactorings on test code quality. We conducted an empirical study [Martins et al. 2025] involving the analysis of refactorings carried out by developers, mined from open-source projects.

RQ$_3$. *How accurately can we suggest test refactoring operations for fixing test smells using ML techniques?*

In **RQ**$_3$, our objective was to explore the performance of machine learning (ML) algorithms in classifying developers' intentions regarding test refactoring. This investigation encompassed two classification tasks: i) classifying code changes where developers would apply refactorings; ii) classifying the specific test refactoring operations.

3. Approach Design

Our three-step approach design aimed at addressing our research problem. We selected subject systems, identified variables, and conducted studies for each RQ. Figure 1 illustrates each step, described as follows.

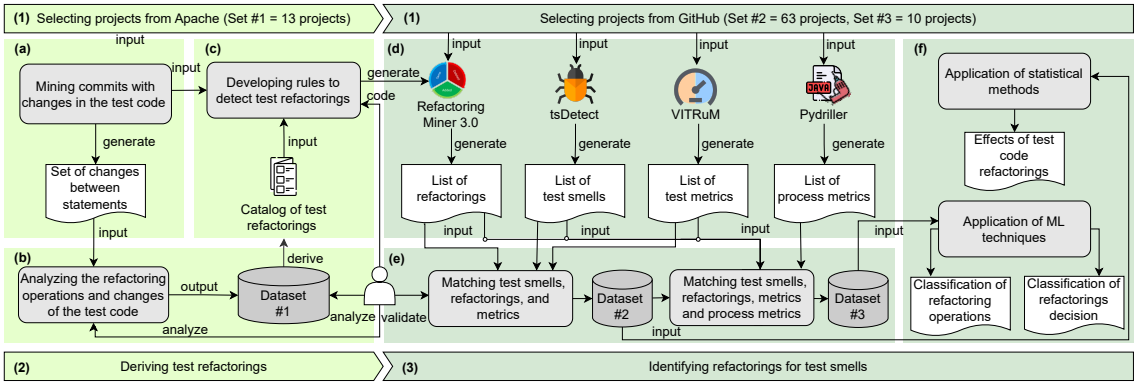

Figure 1. Overview of our approach.

(1) Selecting subject projects: This step consisted of selecting software projects from GitHub. We chose diverse open-source JAVA projects with tests written using the JUNIT testing framework. *Set #1* comprises 13 open-source JAVA projects from the Apache Foundation, and we used it to perform a manual analysis of test code. *Set #2* comprises 63 projects from GitHub. It was used for repository mining to gather data on test smells, refactorings, and structural metrics using automated tools. *Set #3* comprises ten projects from GitHub, with information on test smells, refactorings, structural and process metrics. It was used to train and evaluate machine learning models.

(2) Deriving test refactorings: This step consisted of extracting test code changes from *Set #1* of projects (step a) and manually classifying them into test smells and test refactoring operations (step b), resulting in *Dataset #1*. The outcome of this process is a catalog describing the reengineering process required to perform test refactoring (step c). This catalog proved instrumental in developing rules to enhance existing repository mining tools for detecting test refactorings (refer to Section 4 – **RQ**$_1$).

(3) Identifying refactoring operations for test smells: This step consisted in collecting data using automated tools (step d). In more detail, we collected information on test smells, test refactoring operations, and structural metrics from *Set #2*, generating *Dataset #2* (step e). In addition, it involved the collection of process metrics from *Set #3* to generate *Dataset #3* (step e). Finally, we analyzed the data from both datasets (step f). We utilized *Dataset #2* to analyze the impacts of test code refactorings on test code quality (refer to Section 5 – **RQ**$_2$), and *Dataset #3* to identify refactoring opportunities and specific operations developers would undertake in the test code (refer to Section 6 – **RQ**$_3$). It is worth noting that each dataset was used to investigate a specific research question and, therefore, offers complementary insights for our analysis.

4. Mining Test Refactorings in Practice (RQ1)

Considering the gap between the research on test code refactoring and its adoption in practice, we argue analyzing the refactorings performed by developers may provide insights into the problems they commonly face and the current development practices they use to refactor the test code.

This section presents a summary of our empirical study [Martins et al. 2024a] to answer $\mathbf{RQ}_1$. The *goal* of our empirical study is to analyze the test refactoring operations performed by developers in practice, with the *purpose* of creating a catalog of the test smells developers deem as problematic and which test refactoring operations developers apply to fix the test smells. The *perspective* is both researchers and practitioners who are interested in the problems affecting the test code and how to solve them.

In order to perform such investigation, we analyzed the change history of 375 test files across 13 open-source JAVA projects with test cases written with either JUNIT4, JUNIT5, or both over three years to catalog the refactoring operations performed in the test code to address test smells. We analyzed the changes applied to a version v_n of a test class to classify the refactoring operations applied by developers and its previous version v_{n-1} to classify the test smell fixed through the refactoring operation. After classifying test code modifications, we presented the TSR-Catalog, a catalog for Test Smells Refactorings [Martins et al. 2023d], outlining the reengineering process for conducting test-specific refactorings and leveraging the identified patterns.

Although a recent literature review reports more than 180 test smells [Garousi and Küçük 2018], we found only nine test smells occurring in the 13 open-source projects. In particular, two test-specific refactorings are associated with the upgrade of JUNIT version targeting two test smells (*Handling Exception* and *Assertion Roulette*). Other nine version-agnostic refactorings were identified for seven test smells (*Inappropriate Assertion*, *Assertion Roulette*, *Bad Naming*, *Ignored Test*, *Empty Test*, *Unknown Test*, and *Sleepy Test*). While many of these test-specific refactorings are documented in the literature, this analysis unveils new test-specific refactorings aimed at dealing with the *Inappropriate Assertion* test smell, e.g., developers misuse the *assertEquals* to verify whether a method call returns true instead of using *assertTrue*. In addition, the *Handling Exception* is the most prevalent test smell, with developers frequently replacing *try/catch* blocks and outdated JUNIT annotations with *assertThrows*.

> **Summary**$_1$. *We identified and cataloged 9 test smells that could be fixed by 11 test-specific refactorings. In addition, developers' feedback showed appreciation for the efforts to improve test quality.*

To validate the proposed catalog, we sought feedback from software developers who have contributed to the projects included in our dataset to allow gathering their perspectives on test smells and refactorings. We submitted 13 pull requests to refactor 143 test smells. As a result, developers' feedback suggests a general appreciation for the efforts to improve test quality. In addition, the developers often provided justifications not only for why certain test smells existed but also for why certain refactorings were not worth the change, which offered historical or functional insights that might take time to be apparent from data analysis. Moreover, those discussions often led to further improve-

ments and refinements to the changes beyond our initial suggestions, which refactored more instances of certain test smells in some projects.

In summary, analyzing test refactoring operations in $\mathbf{RQ}_1$ helped reveal (a) the actual issues deemed by developers as test smells and (b) how to fix such issues (i.e., refactor them) in practice. To sum up, this RQ presents the following contributions:

- A manually curated dataset [Martins et al. 2023c] of test-specific refactorings based on 375 test files, showing how developers refactor test code in practice;
- A catalog, publicly available [Martins et al. 2023d], listing the test smells observed in practice alongside their corresponding test-specific refactorings, derived from both the literature and practice;
- A qualitative analysis of the acceptance of our dataset based on 13 pull requests refactoring 143 test smells, reaching a 77% acceptance rate.

5. Test Refactorings Impact on Test Code Quality (RQ2)

Over the last decades, researchers identified both benefits and drawbacks of the application refactorings with of automated recommenders, finding that, while refactoring is theoretically associated with modifications that do not affect the external behavior of source code, it can induce defects [Bavota et al. 2015] or smells [Tufano et al. 2017]. While previous work investigated production code refactoring, there is a lack of investigation into how refactoring is applied to test code. In addition, we do not know whether similar effects can arise when performing test refactoring.

This section addresses this knowledge gap by summarizing our exploratory empirical study [Martins et al. 2025] to answer $\mathbf{RQ}_2$. The *goal* of the empirical study is 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, and (2) as a consequence, to what extent test refactoring operations are effective in improving quality of test classes. In other terms, we are first interested in assessing the *quantity* of test refactoring operations performed on classes exhibiting test code quality issues and, in the second place, the *quality* of the test refactoring operations applied in terms of improvements provided to test code quality. The *perspective* is researchers and practitioners interested in understanding the relationship and effects of test refactoring operations on the quality of test classes.

In order to perform such investigation, we collected test refactoring data from the change history of 63 open-source JAVA projects from GitHub. Next, we combined them with data from automated instruments able to profile test code from the perspective of quality metrics and information on test smells. In particular, we extended state-of-the-art tools to extract 13 test refactoring using TESTREFACTORINGMINER [Tsantalis et al. 2013, Martins et al. 2023b], six test smells using TSDETECT [Peruma et al. 2020a], and six quality metrics calculated with VITRUM [Pecorelli et al. 2020b].

First, we are interested in assessing whether refactoring operations are more likely to be observed on test classes exhibiting test code quality concerns. As such, our *dependent variables* are refactoring operations (of different types) observed across releases,

and our *independent variables* are the (i) test code quality metrics and (ii) presence of test smells (of different types). Second, we are interested in assessing the impact of test refactoring on the test code quality. As such, we swapped independent and dependent variables. Subsequently, we devised a *Logistic Regression Model* for each refactoring operation considered in the study. For each statistical model, we (i) assessed whether each independent variable significantly correlates with the dependent variable (significance level of α = 5%), and (ii) quantified this correlation using the Odds Ratio between independent and dependent variables [Nelder and Wedderburn 1972].

Our first analysis aimed to assess whether refactoring operations are more likely to be observed on test classes exhibiting test code quality concerns. The results show that the *Assertion Density* metric is related to most test refactorings. It indicates that the more assertions in the test code, the more likely developers would refactor it. 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* test smell but no significant correlation exists between them, indicating that those refactorings can stem from variations in coding styles or project-specific guidelines.

> **Summary**$_{2.1}$**.** *Low-quality test classes drive refactorings, in particular those related to Assertion Density metrics. In addition, test smell-driven refactorings lack consistent correlation, suggesting diverse motivations beyond test smell removal.*

Our second analysis aimed at investigating the effects of test refactorings on quality metrics and test smells. The results show that most values remained the same before and after applying test refactorings. However, we have some particular cases. The *Extract Class* refactoring improved the test classes concerning the *Lines of Code*, *Number of Methods*, and *Weight Method Class*. Differently, after applying the *Extract Method* refactoring, the *Number of Methods* and *Weight Method Class* metrics increased their values. These results were quite expected. 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.*

> **Summary**$_{2.2}$**.** *Several structural metrics did not exhibit statistical significance differences before and after applying the test refactorings. Yet, the test code quality had a slight improvement in terms of complexity, coupling, and size when considering the refactorings from Fowler's catalog and some test-specific refactorings (i.e., the Replace Rule w/ assertThrows, Replace Conditional by ParameterizedTest, and Replace @Test with assertThrows refactorings).*

The key findings of **RQ**$_2$ reveal that test refactoring operations typically address low-quality test cases identified by test smells and quality metrics, and it can improve the overall quality of test cases. To sum up, this RQ presents the following contributions:

- An empirical understanding of the factors triggering test refactoring operations, which comprises an analysis of how test code quality come into play;
- Evidence and analysis of the impact of test refactoring on test code quality from the perspective of test smells and structural metrics;

- An online appendix [Martins et al. 2023e] in which we provide all material and scripts employed to address the goals of the study.

6. Developer-Oriented Test Refactoring Recommendations (RQ3)

ML emerges as a promising solution to overcome the need for more tool adoption in software development. ML algorithms can support identifying refactoring opportunities and recommending proper refactorings that align closely with developers' practices. For instance, [Aniche et al. 2022] investigated the effectiveness of ML algorithms to predict refactorings. While these algorithms performed well in predicting refactoring opportunities in production code, it's their findings may not be directly applicable to test code.

This section presents our investigation of the performance of supervised ML algorithms in classifying the developers' intention to apply test refactoring, answering $\mathbf{RQ}_3$. Our *goal* was to investigate the effectiveness of supervised ML algorithms in classifying the code changes where practitioners apply test refactoring actions, considering two levels of granularity: i) the classification of code changes where practitioners apply a refactoring in test classes and ii) the classification of the specific test refactoring action applied by practitioners in a certain code change. Both levels have the *purpose* of studying the feasibility of an automated instrument for test code quality assurance, which practitioners can use to identify test classes requiring maintenance effort and assess how to improve them. The *perspective* is of researchers and practitioners: the former are interested in understanding the performance and limitations of classification models to identify test refactoring opportunities. At the same time, the latter assesses how feasible the proposed automated solution of test code quality assurance would be in practice.

Our empirical investigation slued round two research objectives. As a first step, we explored how accurately supervised ML algorithms classify the code changes where developers would apply test refactoring. We evaluated the performance of six ML algorithms (*Decision Trees*, *Naive Bayes, Logistic Regression, Extra-Tree, Support Vector Machine,* and *Random Forest*) to classify specific test code changes in which practitioners would do test refactoring using test smells, test metrics, and process variables, i.e., code churn, as predictors. After analyzing the performance of ML models in classifying the developer's intention to perform test refactoring, we verified the performance of classifying a specific test code refactoring operation.

To conduct it, we built a dataset encompassing information on 50 different test refactoring operations on ten open-source JAVA projects. In addition, the dataset comprises 21 test smells identified in test classes, 5 test metrics extracted from test classes, and 13 process metrics extracted from both production and test classes. Then, we performed a feature selection to identify the relevant metrics to use as predictors. We quantified the predictive power of each metric in terms of information gain to measure how much a model would benefit from the presence of a metric. At the end, we considered the metrics having an information gain higher than zero as predictors, i.e., we discarded the metrics that did not provide any beneficial effect.

After completing the feature selection, we identified the best ML algorithm for classifying the developers' intentions and test refactoring operations. The literature on test refactoring classification is embryonic; therefore, we took this opportunity to benchmark learning algorithms with different characteristics. To assess the performance of our

models, we performed a walk-forward validation, applying it to individual projects. We decided to apply that validation because we relied on temporal data, so it is crucial to maintain the chronological order to avoid data leakage. In the k-fold cross-validation (widely used), the data is randomly partitioned, which can break the temporal structure.

It is important to point out that test refactoring is an unbalanced problem. The number of test cases refactored instances represents almost 2% of the total amount of test cases on our dataset. As such, the problem was largely underrepresented, threatening the ability of ML algorithms to learn the characteristics of test refactoring properly. For this reason, we tried to improve the performance by experimenting with several under- and over-sampling techniques to balance the data. As for under-sampling, we experimented the advanced *NearMiss 1*, *NearMiss 2*, and *NearMiss 3* algorithms. We also experimented with the *Random Undersampling* approach, which randomly explores the distribution of majority instances and under-samples them. In terms of over-sampling approaches, we experimented with *SMOTE*. In addition, we also experimented with the RANDOM OVERSAMPLING approach, which randomly explores the distribution of the minority class and over-samples them.

We ran and analyzed 540 different models to classify the code changes in test classes. As a result, we found out that the SVM algorithm outperforms the others, with performance varying between 30% to 100%. In particular, four test smells (*Default Test, Empty Test, Verbose Test,* and *Dependent Test*), did not contribute to the models (the predictive power of these metrics was always zero). Then, for most of the test smells in analysis, i.e., *Constructor Initialization, General Fixture, Ignored Test, Mystery Guest, Redundant Assertion, Resource Optimism,* and *Sleepy Test*, the values are condensed around zero, so their contribution to the models is minimal. This result suggests that while some smells can be good indicators to classify test code changes, they did not represent the best dimension to use.

> **Summary**$_{3.1}$**.** *The best model to predict test code changes is Support Vector Machines with Random Undersampling as balancing technique. The F-Measure obtained varies between 30% and 100%. The info gain analysis shows that test code and process metrics have higher contributions to the model than test smells.*

In addition, we analyzed how supervised ML models classify specific test refactoring operations. Before proceeding, it is important to highlight a consideration that reduced the number of refactorings analyzed. First, we ran 540 models for each test refactoring, for a total of 27,000 models. Once we had collected the results, we observed that for 32 refactorings, not enough information was found to train our models due to the low number of true instances. On the remaining 18 refactorings, for 11 we were able to gather information from only one project, for 4 cases from two projects, and for the last three refactorings, i.e., *Modify Method Annotation*, *Remove Method Annotation*, and *Add Method Annotation*, we gathered information from four, six, and seven projects, respectively. Concerning the info gain analysis, we observed a similar trend to that previously reported in the classification of whether refactor the code, but with slightly lower values. In addition, we found out that both under- and over-sampling techniques contribute to ML models except for seven refactorings, i.e., *Extract Attribute*, *Extract Variable*, *Pull Up Attribute*, *Pull Up Method*, *Inline Method*, *Modify Method Annotation*, and *Move Method*.

> **Summary**$_{3.2}$. *When we classify specific test refactoring operations, performance decreases due to a lower amount of data. The F-measure obtained varies between 19% and 60%. The info gain analysis shows that test code and process metrics have higher contributions to the model than test smells.*

The key finding of **RQ**$_3$ shows that using ML to classify test code changes is feasible. Still, the performance decreases when classifying specific test refactoring operations due to the lower amount of data analyzed. The results also shows the need to explore the socio-technical factors influencing the developer's activities. To sum up, the contributions of this RQ are:

- An ML approach to classify the developers' intention to apply test refactoring and the test-specific operations applied;
- A publicly available replication package [Martins et al. 2024b] with data, scripts, and results of our experiment.

7. Related Works

This section compares this work with the most relevant related studies available at the time of writing this paper.

7.1. Investigation of Test Smells Effects on Software Quality

Several studies have explored the relationship between test smells and software quality, revealing their negative impact on code comprehension and maintainability. [Bavota et al. 2012] and [Bavota et al. 2015] conducted empirical studies analyzing the diffusion of test smells and their effects on software maintenance, concluding that test smells reduce test code comprehension. [Peruma et al. 2019] found that test smells are prevalent in Android apps and often appear early in development. Other studies, such as [Pecorelli et al. 2020a] and [Martins et al. 2023a], examined correlations between test smells and structural metrics, indicating that test classes often have low design quality.

In contrast, this study extends current knowledge by assessing how test refactoring is applied and its impact on multiple aspects of test code. We do not limit ourselves to analyzing test smells but also consider additional indicators of test code quality. In this sense, our study presents a more comprehensive analysis of test refactoring (RQ_2).

7.2. Refactorings to Fix Test Smells

A primary line of research on test refactoring has focused on understanding the relationship between refactoring and test smells. [Peruma et al. 2020b] and [Kim et al. 2021] analyzed how refactoring changes affect test smells, finding that refactoring often coincides with test smell removal but may not always be performed for that purpose. A second line of research is represented by qualitative studies targeting the developer's perception of test refactoring. [Soares et al. 2022] investigated developers' awareness and acceptance of test refactorings, highlighting a lack of awareness of test refactoring techniques.

In contrast, we analyzed the actual refactorings developers performed in test code over time, providing a more objective insight into how they tackle test smells. Understanding these developer practices not only enlightens us on the prevalent features regularly employed in test code but also empowers us to propose test refactorings that better align with developers' perspectives and preferences (RQ_1).

7.3. Machine Learning Techniques to Handle Test Smells

Some studies have also investigated the feasibility of using ML techniques to handle test smells. For example, [Martins et al. 2021a] used structural metrics to train ML models for detecting test smells. Similarly, Hadj et al. [Hadj-Kacem and Bouassida 2021] proposed a multi-label classification approach using deep representations of test code. In both studies, Random Forest performed best.

While some studies have used ML techniques to detect test smells, we observe a lack of studies that explicitly consider the classification of test code refactoring operations. The closest work is the one by [Aniche et al. 2022], in which the authors proposed an approach to predict refactoring operations only in production code using process and structural metrics as predictors. Our study addresses this gap by focusing on test code, considering its peculiarities and the evolutionary nature of the test source code—a perspective not explicitly considered in the Aniche's work (RQ_3).

8. Conclusions

This study investigated how developers refactor the test code to fix test smells in practice and what factors drive developers to refactor the test code. In addition, we explored ML techniques to classify whether developers would apply refactorings in the test code and which test-specific refactoring operations they would apply. By considering the developers' perspective, this research lays the foundation for more tailored refactoring techniques that can enhance the effectiveness of test-specific refactoring recommendation tools.

Therefore, we have two main contributions of this work.

1. **A Catalog with common test smells and test refactorings in practice**, compiled through a manual analysis of modified test cases, providing a systematic understanding of how developers address test smells.
2. **Identification of Refactoring Opportunities and Proper Fixes for Test Smells**, achieved by developing ML models to classify refactoring decisions and predict specific refactoring operations, advancing automated test refactoring recommendations.

Beyond these contributions, this research on the detection of test smells and identification of proper refactorings to handle them has had a tangible impact on both academia and software development practice. With 261 citations and an H-index of 9[1], this work has influenced ongoing research in software testing and refactoring, demonstrating its significance in the field. Moreover, we proposed tools that support developers in maintaining high-quality test code, improving software reliability and maintainability.

Future research can expand in several directions. First, the proposed catalog of test smells and refactoring strategies would benefit from empirical validation with developers through surveys and interviews to gather feedback and real-world insights. Additionally, controlled experiments are encouraged to assess the impact of these refactorings on quality attributes such as test reliability, maintainability, and execution time. Another avenue involves generating synthetic data for model training, as the current datasets are limited in the number of refactorings represented; i.e., creating realistic and diverse synthetic data could improve model generalization and robustness. In addition, given that

[1]From Scholar https://scholar.google.com/citations?hl=en&user=pDe4dNsAAAAJ — March 21, 2025.

process metrics significantly contributed to the performance of the ML models, analyzing socio-technical factors in developers' test code refactoring activities could yield valuable insights into adoption and effectiveness. Lastly, exploring supervised machine learning algorithms for test refactoring recommendations – by refining models, experimenting with different algorithms, and incorporating varied project contexts and additional features – could enhance both the accuracy and applicability of automated suggestions.

Acknowledgements

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

References

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. In *Evaluation and Assessment in Software Engineering*, page 170–180, New York, NY, USA. ACM.

Aniche, M., Maziero, E., Durelli, R., and Durelli, V. H. S. (2022). The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance*, pages 56–65, New York, NY, USA. IEEE.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A., and Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094.

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. (2018). Deflaker: Automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*, page 433–444, New York, NY, USA. ACM.

Campos, D., Rocha, L., and Machado, I. (2021). Developers perception on the severity of test smells: an empirical study. In *Iberoamerican Conference on Software Engineering*, pages 1–14, Costa Rica. arxiv.

Deursen, A., Moonen, L. M., Bergh, A., and Kok, G. (2001). Refactoring test code. Technical report, Centre for Mathematics and Computer Science, NLD.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, Upper Saddle River, NJ.

Garousi, V. and Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81.

Hadj-Kacem, M. and Bouassida, N. (2021). A multi-label classification approach for detecting test smells over java projects. *Journal of King Saud University-Computer and Information Sciences*.

Kim, D. J., Chen, T.-H. P., and Yang, J. (2021). The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):1–47.

Martins, L., Bezerra, C., Costa, H., and Machado, I. (2021a). Smart prediction for refactorings in the software test code. In *Brazilian Symposium on Software Engineering*, page 115–120, New York, NY, USA. ACM.

Martins, L., Brito, V., Feitosa, D., Rocha, L., Costa, H., and Machado, I. (2021b). From blackboard to the office: A look into how practitioners perceive software testing education. In *Evaluation and Assessment in Software Engineering*, page 211–220, New York, NY, USA. ACM.

Martins, L., Costa, H., and Machado, I. (2023a). On the diffusion of test smells and their relationship with test code quality of java projects. *Journal of Software: Evolution and Process*, page e2532.

Martins, L., Costa, H., Ribeiro, M., Palomba, F., and Machado, I. (2023b). Automating test-specific refactoring mining: A mixed-method investigation. In *Proceedings of the 23rd IEEE Int.l Working Conf. on Source Code Analysis and Manipulation*, page 12, Los Alamitos, CA, USA. IEEE Computer Society.

Martins, L., Ghaleb, T., Costa, H., and Machado, I. (2023c). Curated dataset of test-specific refactorings. Available at https://figshare.com/s/3cd337c00ba36954854e.

Martins, L., Ghaleb, T., Costa, H., and Machado, I. (2023d). Tsr-catalog: The catalog of test smells refactorings. Available at https://tsr-catalog.readthedocs.io/en/latest/.

Martins, L., Ghaleb, T. A., Costa, H., and Machado, I. (2024a). A comprehensive catalog of refactoring strategies to handle test smells in java-based systems. *Software Quality Journal*, 32(2):641–679.

Martins, L., Pontillo, V., Costa, H., Ferrucci, F., Palomba, F., and Machado, I. (2024b). Toward classifying test refactoring opportunities using supervised machine learning: How far can we go? — online appendix. Available at https://figshare.com/s/f1bfa5fdbf1d4caf1d27.

Martins, L., Pontillo, V., Costa, H., Ferrucci, F., Palomba, F., and Machado, I. (2025). Test code refactoring unveiled: where and how does it affect test code quality and effectiveness? *Empirical Software Engineering*, 30(1):1–39.

Martins, L., Pontilo, V., Costa, H., Palomba, F., and Machado, I. (2023e). Online appendix — Test code refactoring unveiled: where and how does it affect test code quality and effectiveness? Available at https://figshare.com/s/2f1d6dc0134f5a95d745.

Melo, S., Moreira, V., Paschoal, L. N., and Souza, S. (2020). Testing education: A survey on a global scale. In *In 34th Brazilian Symposium on Software Engineering*, page 554–563, New York, NY, USA. ACM.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Addison-Wesley Signature Series. Addison-Wesley, Upper Saddle River, NJ.

Nelder, J. A. and Wedderburn, R. W. (1972). Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384.

Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R., and De Lucia, A. (2016). On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, page 5–14, New York, NY, USA. ACM.

Pecorelli, F., Catolino, G., Ferrucci, F., De Lucia, A., and Palomba, F. (2020a). Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *Proceedings of the 28th International Conference on Program Comprehension*, page 296–307, New York, NY, USA. ACM.

Pecorelli, F., Di Lillo, G., Palomba, F., and De Lucia, A. (2020b). Vitrum: A plug-in for the visualization of test-related metrics. In *Proceedings of the International Conference on Advanced Visual Interfaces*, New York, NY, USA. ACM.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019). On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, page 193–202, USA. IBM Corp.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020a). Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1650–1654, NY, USA. ACM.

Peruma, A., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020b). An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 350–357, New York, NY, USA. IEEE.

Peruma, A., Simmons, S., AlOmar, E. A., Newman, C. D., Mkaouer, M. W., and Ouni, A. (2022). How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43.

Silva Junior, N., Soares, L. R., Martins, L., and Machado, I. (2020). A survey on test practitioners' awareness of test smells. *CoRR*, abs/2003.05613:1–14.

Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., Fernandes, L., Garcia, A., Fonseca, B., and Santos, A. (2020). Refactoring test smells: A perspective from open-source developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST 20, page 50–59, New York, NY, USA. ACM.

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. M. (2022). Refactoring test smells with junit 5: Why should developers keep up-to-date. *IEEE Transactions on Software Engineering*, pages 1–1.

Spadini, D., Schvarcbacher, M., Oprescu, A.-M., Bruntink, M., and Bacchelli, A. (2020). Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, page 311–321, NY, USA. ACM.

Terragni, V., Salza, P., and Pezze, M. (2020). Measuring software testability modulo test quality. In *Proceedings of the 28th International Conference on Program Comprehension*, page 241–251, New York, NY, USA. ACM.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conf. of the Center for Advanced Studies on Collaborative Research*, page 132–146, USA. IBM Corp.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, page 4–15, New York, NY, USA. ACM.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2017). There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838.