# How Does Test Code Differ From Production Code in Terms of Refactoring? An Empirical Study

Kosei Horikawa*, Yutaro Kashiwa*, Bin Lin†, Kenji Fujiwara‡, Hajimu Iida*
*Nara Institute of Science and Technology, Japan
†Hangzhou Dianzi University, China
‡Nara Women's University, Japan

***Abstract*—Refactoring is a widely applied practice for improving the internal structure of source code without altering its external behavior. Researchers have proposed approaches to detect refactoring operations and investigated their impact on the code quality. However, these studies often focus on production code, paying little attention to test code. It is still unclear whether developers perform refactoring on test code in the same way or for the same purpose. To fill this gap, we first investigate the types and prevalence of refactoring applied in production and test code, and then examine whether these refactorings impact the code quality in a different way. Our results show that certain refactorings are less common in the test code. Besides, while refactoring-related changes in production and/or test code improved readability, they had limited impact on most design smells. We also find that some specific refactoring types do impact certain design smells. These findings indicate the special attention needed for test code when analyzing refactorings.**

***Index Terms*—Refactoring, Code metrics, Code readability, Code smell, Mining software repositories**

## I. Introduction

Refactoring is a well-established practice in software engineering, aimed at improving the internal structure of code without altering its external behavior [1]. Its crucial role for maintaining and enhancing the quality of software systems has been demonstrated over time [2], [3]. Studies have found that refactoring has a positive impact on code design [4], [5], maintainability [6], [7], readability [3], [8], and the prevention of defects [9]. These studies mainly focus on production code. However, it is worth noting that refactoring is also applied to test code. Test code is essential for verifying the correctness and reliability of production code, and it also benefits from refactoring [10]. Several recent studies have investigated the significance of refactoring in test code [11]. Peruma *et al.* [12] have found that refactoring test code can effectively remove test smells — poor practices that hinder the effectiveness of testing [10]. Martins *et al.* [13] found that the coupling, cohesion, and size of the test code improve after certain refactorings while the number of test smells increases.

Several studies have reported that refactoring types differ between production and test code [12], [14]. They claim that developers tend to perform structural changes (*e.g.,*"Move Class") on production code, while test code often undergoes naming-related modifications (*e.g.,*"Rename Method"). Moreover, the refactorings on production code are mainly for reducing complexity, while those in test code focus on readability. These studies do not empirically measure readability or design smells, therefore, it remains unclear whether such refactorings actually impact these quality attributes.

This study aims to bridge this gap by investigating the refactoring activities performed in both production and test code. More specifically, we first examine what types of refactorings are more commonly applied and compared the differences between production and test code. With the dataset encompassing 455 projects with a total of 2,016,632 commits, we observe 6,678,593 refactorings, of which 1,177,833 are applied to test code and 5,500,760 to production code. Interestingly, "Change Variable Type" and "Add Method Annotation" refactorings are much more common in test code.

We then analyze the impact of these refactorings on code quality, particularly readability and design smells, in both production and test code. Our findings indicate that code changes that refactor production code and test code do not have the same impact on readability while they both improve it. For example, test code refactoring is more likely to simplify the test code (*e.g.,* improve readability of identifiers), but production code refactoring focuses on structural improvements. In terms of the design smells, our results show that refactorings had little impact on most design smells, indicating that removing design smells is not a primary goal of day-to-day refactoring. These results indicate that developers refactor test code in a different way and for different purposes.

**Replication Package:** To facilitate replication and further studies, we provide the data used in our replication package.[1]

## II. Research Questions

Our goal is to reveal the differences in refactoring operations performed on production and test code and to disclose the difference in their impacts on the quality of code. We propose the following two research questions (RQs).

**$RQ_1$: What kinds of refactorings are applied to test code and production code?**

While previous studies have investigated the types of refactoring applied in test code [11], [15], they both analyzed a very limited number of systems and refactoring types. In this RQ, we aim to analyze a large number of projects and apply a state-of-the-art refactoring tool to detect a wide range of refactoring types. Moreover, we categorize refactoring operations based on which part of the code (only production code, only test code, or both) these refactorings are applied to.

[1]https://github.com/Mont9165/ProdTestRefactoringAndMetrics

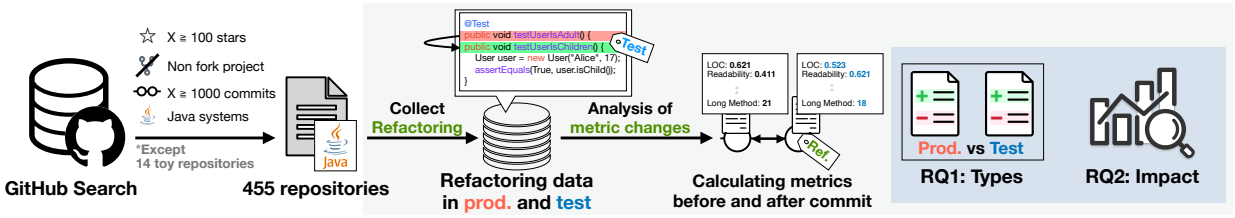


Fig. 1: Overview of the Data Analysis

***$RQ_2$*: How do refactoring-related code changes impact the code quality of test code and production code?**

Lima *et al.* [16] conducted interviews to investigate the purposes of refactoring test code. As a result, 75% of interviewees answered "*supporting automated code maintenance*" as a key benefit of refactoring, while 60% highlighted "*improving code readability*". As code maintenance typically aims to improve code quality and over half of the interviewees mentioned readability, we would like to investigate how code quality metrics and readability are impacted by refactorings and whether these impacts differ between test code and production code.

## III. Study Design

### A. Context Selection

We selected as context for our study 455 projects from GitHub, satisfying the following criteria, using the SEART GitHub Search Engine [17].[2]

- **Java systems.** In this study, we use a state-of-the-art refactoring tool [18] that only works on Java code.
- **Active and well-developed projects.** We excluded inactive projects that did not have a commit between March 1, 2024, and September 1, 2024. Also, we excluded projects that do not have more than 1,000 commits in their commit history to ensure a sufficient number of refactorings.
- **Popular projects.** We selected projects with at least 100 forks and 100 stars. These metrics serve as proxies for popularity and help eliminate toy projects.
- **Non-forked projects.** We excluded forked projects.

We identified 469 projects that met our selection criteria. We also manually examined these projects to exclude two tutorial repositories. Furthermore, we excluded 11 projects due to excessive time needed for refactoring detection and metrics calculation, and 1 project because no refactoring was detected.

### B. Data Analysis

To answer our RQs, we first detect refactorings from the selected repositories, and then assess the changes in readability and code quality metrics before and after each refactoring commit. The detailed methodology is described as follows.

*1) Refactoring Prevalence (RQ1):* We utilized RefactoringMiner 3.0.0[3], the state-of-the-art refactoring detection tool, to identify refactorings in the studied repositories. This tool extracts refactoring operations by analyzing two consecutive commits using an Abstract Syntax Tree-based matching algorithm. It supports the detection of 99 different refactoring types and is reported to have an overall F1-score of 99.5% [18]. When extracting refactoring operations, we also retrieved the paths of the Java files where the refactoring operations were applied. Using these file paths, we determined whether the refactorings were applied to production code only, test code only, or both. The heuristic behind this classification is: if the file path contains "`test`", then the given file is part of test code, otherwise it is production code.[4] Finally, to identify the characteristics of refactoring on production code and test code, we compared the following metrics: *the number of commits, the number of files that were refactored, the number of refactoring instances, and the proportion of refactoring types*. As commits can modify production code and test code simultaneously, we classify them into three groups: commits that modify only production code (*i.e.,* production commits), commits that modify only test code (*i.e.,* test commits), and commits that modify both (*i.e.,* co-occurring commits).

*2) Impact on Code Quality and Readability (RQ2):* After identifying the refactored classes in production and test code in RQ1, we measured the changes in the code quality of these classes after the refactoring. To measure the code quality, we utilized a readability measurement tool and a software quality measurement tool. The readability measurement tool, developed by Scalabrino *et al.* [19], assesses code readability using structural features (*e.g.,* line length, indentation, number of identifiers) and textual features (*e.g.,* comment consistency, vocabulary overlap, readability indices).

For code quality, we employed DesigniteJava[5], developed by Sharma [20], which calculates various code metrics and detects a wide variety of code smells. The tool can measure coupling metrics (*e.g., FANIN*, *FANOUT*), inheritance metrics (*e.g., Depth of Inheritance Tree*, *Number of Children*), size and complexity metrics (*e.g., Lines of Code*, *Weighted Methods per Class*), cohesion metrics (*e.g., Lack of Cohesion in Methods*), and structural metrics (*e.g., Number of Fields*, *Number of Methods*). We also use this tool to detect design smells (*e.g., Broken Modularization*, *Deep Hierarchy*) to verify the claim by the previous study [14] that developers are more likely to perform structural changes on production code. By comparing how design smell changes after refactoring, we can understand the impact on code sturucture.

[2]https://seart-ghs.si.usi.ch/
[3]https://github.com/tsantalis/RefactoringMiner/tree/3.0.0
[4]Lowercase file paths are used.
[5]https://github.com/tushartushar/DesigniteJava

TABLE I: Summary of Refactoring Frequency Across Production and Test Code

| Type | Refactoring commits | | | Refactored files | | Refactoring instances | |
|---|---|---|---|---|---|---|---|
| | Prod. commits | Test commits | Co-occur. commits | Prod. code | Test code | Prod. code | Test code |
| Mean | 793 | 114 | 130 | 1,321 | 378 | 12,090 | 2,589 |
| Median | 438 | 43 | 47 | 651 | 123 | 5,170 | 641 |
| Sum | 360,655 (76.4%) | 52,068 (11.0%) | 59,180 (12.5%) | 601,278 (77.7%) | 172,195 (22.3%) | 5,500,760 (82.4%) | 1,177,833 (17.6%) |

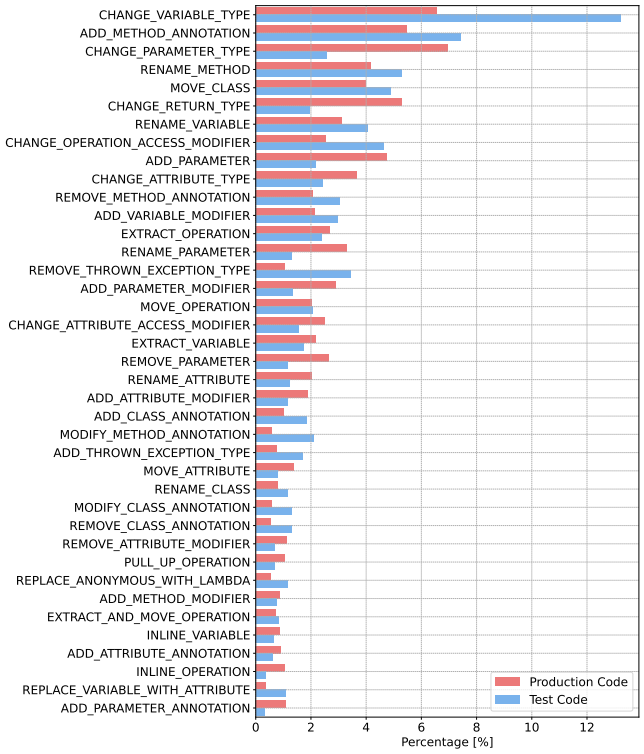


Fig. 2: Comparison of Refactoring Types in Production vs. Test Code

## IV. Results

### A. RQ1 (Refactoring Prevalence)

**Frequency.** We applied RefactoringMiner to a total of 2,016,632 commits from 455 projects and identified refactoring operations in 471,903 (23.4%) commits. Table I lists the number of commits and files containing refactoring, as well as the number of identified refactorings. Overall, 76.4% commits modified only the production code, 11.0% modified only the test code, and 12.5% modified both the production and test code. While previous studies [21] have shown that developers often maintain test code alongside production code, our results suggest that developers also dedicate significant efforts to test-code-focused maintenance.

Regarding the refactoring-related files, we found that over 77% of the files are production code. Moreover, the number of identified refactorings in production code is around four times of that in test code. Both results indicate that refactoring on production code is much more than test code refactoring.

**Types.** Figure 2 illustrates the percentage of refactoring operations performed on the production code and test code. The five most common types of refactorings in production commits are "Change Parameter Type" (7.0%), "Change Variable Type" (6.6%), "Add Method Annotation" (5.5%), "Change Return Type" (5.3%), and "Add Parameter" (4.7%). Interestingly, three out of these five refactoring types are API-related changes (*i.e.,*"Change Parameter Type", "Change Return Type", and "Add Parameter"). To identify the refactorings unique to production code, we compared the ratio of different types of refactoring operations in production code to that in test code. The biggest differences go to the following refactoring types: "Split Conditional" (13.3 times), "Merge Conditional" (11.4 times), "Encapsulate Attribute" (6.3 times), "Change Type Declaration Kind" (5.5 times), "Extract Interface" (5.4 times). This indicates that operations involving modifying conditions and attributes are more likely to be performed in production code.

As for the test code, the five most common types of refactorings are "Change Variable Type" (13.2%), "Add Method Annotation" (7.4%), "Rename Method" (5.3%), "Move Class" (4.9%), and "Change Operation Access Modifier" (4.6%). Similarly, we also identified some refactoring types unique to test code: "Assert Throws" (3317.0 times)[6], "Parameterize Test" (70.1 times), "Move Code" (5.1 times), "Modify Method Annotation" (3.6 times), "Split Operation" (3.6 times). These results suggest that method-level refactoring is more likely to be performed in test code.

> **Answer to RQ1.** Refactoring activities are significantly more prevalent in production code than in test code, both in frequency and diversity of operations. Production code refactorings often involve structural and API-level changes, while test code refactorings are more focused on operations like renaming methods, moving classes, and test-specific transformations.

### B. RQ2 (Refactoring Impact)

Here we discuss the impact of refactoring on code metrics, design smells, and readability. Below we describe the results of these metrics with statistically significant differences.

[6]Since "Assert Throws" and "Parameterize Test" are test-specific refactorings that RefactoringMiner can detect, the significant differences between production and test code are not surprising. However, RefactoringMiner produced several false positives when detecting these types in the production code so the values are not infinite (*i.e.,* divided by non-zero).

TABLE II: Code Metric Changes in Production and Test Code

| Cat. | Metrics | Prod. | Test |
|---|---|---|---|
| Size | Lines of Code | -5.42 | +3.02 |
| Size | Number of Fields | -0.40 | +0.42 |
| Size | Number of Methods | -0.77 | +0.74 |
| Size | Number of Public Methods | -0.12 | +0.22 |
| Complexity | Weighted Methods per Class | -1.62 | +2.07 |
| Coupling | FANIN | -0.77 | +1.31 |
| Coupling | FANOUT | +0.05 | -0.41 |
| Coupling | Number of Public Fields | -0.13 | +0.22 |
| Cohesion | Lack of Cohesion in Methods | +0.02 | -0.03 |
| Inheritance | Depth of Inheritance Tree | +0.05 | -0.01 |
| Inheritance | Number of Children | -0.04 | +0.05 |

**Code metrics.** Table II summarizes the changes in code metrics after refactoring. The table only shows the metrics with statistically significant differences between the production code and test code, detected by the Wilcoxon signed-rank test ($\alpha$ = 0.01). Interestingly, we observed that many metrics regarding the Size category in production code decreased while they increased in test code. For example, *Lines of Code* metric decreased by an average of 5.42 lines in the production code, whereas they increased by 3.02 lines in the test code. When looking into the refactoring types that impact *Lines of Code*, "PUSH DOWN OPERATION" and "EXTRACT CLASS" reduce 69.7 and 62.0 lines of code on average in the production code. "MOVE SOURCE FOLDER" and "EXTRACT OPERATION" increase 73.6 and 14.0 lines in the production code. Similarly, the Complexity metric *Weighted Methods per Class* (the sum of the complexities of the methods in the target class) also increases for test code and decreases for production code.

In the Coupling category, the metrics *FANIN* (*i.e.,* the number of other modules or functions that invoke or depend on a given module) and *FANOUT* (*i.e.,* the number of modules or functions that a given module invokes or depends on) exhibit contrasting trends. After refactoring, *FANIN* decreases in production code but increases in test code, whereas *FANOUT* increases in production code and decreases in test code. For production code, it means improved modularity and separation of concerns, as lower *FANIN* implies that individual modules are less central or less relied upon by others. Conversely, the increase in *FANOUT* indicates that modules are now delegating more responsibilities, possibly to newly introduced utility components, which aligns with principles of responsibility-driven design. In contrast, the increase in *FANIN* and decrease in *FANOUT* in test code suggest a shift toward more focused and isolated testing. Higher *FANIN* in test code implies that more test cases are targeting specific production modules, potentially reflecting improved test coverage or granularity. These trends imply that the refactoring process has led to a more modular and testable architecture.

For the Inheritance category, *Depth of Inheritance Tree* increases in production code and decreases in test code, while *Number of Children* exhibits an opposite trend. This suggests that the production code shifts toward deeper but narrower class hierarchies, potentially reflecting more specialized design with reduced subclassing. In contrast, the opposite trend in test code indicates a move toward shared base test classes to promote reuse and simplify test maintenance.

TABLE III: Design Smell Changes in Prod. and Test Code

| Design Smell | Prod. | Test |
|---|---|---|
| Unutilized Abstraction | +0.07 | -0.22 |
| Deficient Encapsulation | -0.01 | +0.02 |
| Cyclic-Dependent Modularization | -0.01 | +0.02 |
| Insufficient Modularization | -0.01 | +0.02 |
| Broken Hierarchy | +0.00 | -0.01 |

TABLE IV: Readability Changes In Production and Test Code

| Metrics | Prod. | Test |
|---|---|---|
| Synonym Commented Words AVG | +14.5% | +1.9% |
| Semantic Text Coherence Normalized | +0.5% | +0.9% |
| Number of Senses AVG | +0.4% | -0.7% |
| Method Chains AVG | +0.5% | +0.6% |
| Comments Readability | +3.8% | +2.3% |
| Commented Words AVG | +102.1% | +21.6% |
| Abstractness Words AVG | +0.6% | -1.1% |

**Design smell.** Table III summarizes the changes in design smell after refactoring. Overall, refactoring had a minimal impact on most design smells, suggesting that design smell removal is not a primary goal of day-to-day refactoring. However, our analysis reveals that specific refactoring types can have a significant effect on certain smells. In production code, we observed that targeted refactorings often aim to remedy specific design issues. For instance, structural refactorings effectively reduced modularity-related smells: "MOVE PACKAGE" and "RENAME PACKAGE" decreased *Unutilized Abstraction*, while "EXTRACT CLASS" and "EXTRACT INTERFACE" reduced *Insufficient Modularization* and *Cyclic-Dependent Modularization*, respectively. Similarly, "REMOVE ATTRIBUTE MODIFIER" directly addressed and reduced *Deficient Encapsulation*. These findings show a clear pattern where developers use specific structural refactorings to improve design quality in production code. In test code, organizational changes appeared to introduce smells. "MOVE SOURCE FOLDER" increases both *Deficient Encapsulation* and *Insufficient Modularization*, and "RENAME PACKAGE" increases *Cyclic-Dependent Modularization*. Moreover, some refactorings had counter-intuitive effects even in production code; "EEXTRACT SUPERCLASS" was found to increase *Broken Hierarchy*, suggesting that creating new abstractions can be detrimental if not done carefully.

**Readability.** Table IV presents readability metric changes between production and test code, highlighting significant differences. In production code, refactorings primarily enhanced overall consistency and coherence. For example, *Commented Words AVG* improved significantly more in production code (+102.1%) than in test code (+21.6%). Conversely, refactorings in test code focused on improving the clarity of individual test cases through the use of specific and unambiguous identifiers. For *Number of Senses AVG* and *Abstractness Words AVG*, the metrics increase in production code but slightly decline in test code. These contrasting patterns suggest that developers apply readability-enhancing refactorings with different goals:

emphasizing consistency and maintainability in production code, while prioritizing clarity in test code.

**Answer to RQ2.** In production code, refactoring improves structural quality by improve code metrics and addressing specific design smells, with a focus on overall consistency. In contrast, test code refactoring prioritizes clarity and effectiveness, which often worsens code metrics as a side-effect while improving readability through more specific identifiers.

## V. Future Research Direction

**Variety of metrics.** We will integrate more metrics that targeting either production or test code, such as maintainability, implementation smells and testability smells. We will perform finer-grained analysis (*e.g.,* method-level) to reveal how refactorings impact these metrics.

**Interview study.** We will study how developers apply test-specific refactorings in practice. More specifically, we would like to understand the approaches they take, the challenges they face, and the support they need for test refactoring.

**Automatic refactoring based on the different goal.** We would like to build a refactoring recommendation tool which clearly lists potential impact of refactoring and can recommend different types of refactorings based on the context.

## VI. Related work

### A. Refactoring for Production Code and Its Impact

Kim *et al.* [9] conducted a survey at Microsoft to investigate the practical challenges and benefits of refactoring. Their findings revealed that refactoring is perceived to improve software quality and developer productivity. Silva *et al.* [22] examined the motivations behind refactoring activities and found that refactoring is primarily driven by changes in requirements rather than the presence of code smells.

Lin *et al.* [23] analyzed how refactoring impacts the code naturalness (*i.e.,* the nature of code being repetitive and predictable) and found "Extract Method" refactoring tends to improve naturalness, while "Pull Up Method" is the opposite. Bavota *et al.* [2] empirically investigated the relationship between refactoring and quality metrics, as well as code smells. Their findings indicate that quality metrics have little correlation with refactoring.

### B. Comparison of refactoring production code and test code

Tsantalis *et al.* [15] compared the types of refactorings performed to test and production code in three OSS projects. The results show that only three types of refactoring are applied in the test code while 11 types are observed in the production code. Despite the similarity to our RQ1, this study examined only three projects, and the study was conducted in 2013 with limited types of refactorings supported. Peruma *et al.* [12] investigated the refactoring activities of test files and Production files in 250 Android applications. They revealed that developers often perform different types of refactorings on test files, such as "Rename Method" and "Change Variable Type," which are less common in production code. Futhermore, AlOmar *et al.* [14] also investigated both production code and test refactoring. They found that there were many name changes in the test code, while many structural improvements were done in the production code. These survey results were the driving force behind our analysis of readability.

Martins *et al.* [13] investigated the impact of refactoring on test code quality and effectiveness. They found that test refactorings target low-quality test code, but do not strongly correlate with code/mutation coverage metrics. Additionally, refactorings such as "Extract Class" improve the test code's coupling, cohesion, and size. Our work focused on a different set of quality metrics, and their study did not differentiate production code and test code.

## VII. Threats to validity

*Threats to internal validity* concern the reliability of the tool we used to detect refactorings. While RefactoringMiner 3.0.0 and Designite have a high accuracy, it is still possible that some refactorings or code smells are missed out by the tools.

*Threats to construct validity* concern the types of refactoring we investigated. RefactoringMiner 3.0.0 was mainly designed to detect refactoring operations in the production code. We investigated the impact of these refactorings on readability and code metrics. However, it is unclear how some specific test refactorings [24], [25] would impact these metrics. The main goal of our study is to highlight the different purposes behind production code and test code refactoring.

*Threats to external validity* concern the projects we used in the study. Our work collected 455 projects from GitHub. While we have analyzed more projects than existing studies [11], [12], [15], it is still unclear whether analyzing the remaining projects will lead to the same results. Moreover, closed-source projects were not included in our study.

## VIII. Conclusion

We analyzed 6,678,593 refactorings and their impacts on code quality with code metrics, design smell and readability. Our findings show that refactoring-related changes in production and test code improved readability, they do have differences regarding refactoring types and improvement aspects. Our future work would focus on integrating more metrics, conducting interview studies to understand the practice of test refactoring, and proposing a context-aware refactoring tool.

## Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP24K02921, JP24K02923, JP25K21359), as well as JST for the PRESTO grant (JPMJPR22P3), the ASPIRE grant (JPMJAP2415), and the AIP Accelerated Program (JPMJCR25U7).

## REFERENCES


[1] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999.

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

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

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

[5] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *Proceedings of the 9th International Conference on Software Reuse (ICSR)*, vol. 4039, 2006, pp. 287–297.

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

[7] E. A. AlOmar, H. Alrubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 348–357.

[8] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, no. 1, p. 11, 2022.

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

[10] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.

[11] N. A. Nagy and R. Abdalkareem, "On the co-occurrence of refactoring of test and source code," in *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2022, pp. 122–126.

[12] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "An exploratory study on the refactoring of unit test files in android applications," in *Proceedings of the 42nd International Conference on Software Engineering, Workshops (ICSEW)*, 2020, pp. 350–357.

[13] L. A. Martins, V. Pontillo, H. A. X. Costa, F. Ferrucci, F. Palomba, and I. do Carmo Machado, "Test code refactoring unveiled: where and how does it affect test code quality and effectiveness?" *Empir. Softw. Eng.*, vol. 30, no. 1, p. 27, 2025.

[14] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Syst. Appl.*, vol. 167, p. 114176, 2021.

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

[16] D. L. Lima, R. E. de Souza Santos, G. P. Garcia, S. S. da Silva, C. França, and L. F. Capretz, "Software testing and code refactoring: A survey with practitioners," in *Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 500–507.

[17] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 560–564.

[18] P. Alikhanifard and N. Tsantalis, "A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, pp. 40:1–40:63, 2025.

[19] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, 2018.

[20] T. Sharma, "Multi-faceted code smell detection at scale using designitejava 2.0," in *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2024, pp. 284–288.

[21] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 35–46.

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

[23] B. Lin, C. Nagy, G. Bavota, and M. Lanza, "On the impact of refactoring operations on code naturalness," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 594–598.

[24] L. A. Martins, H. A. X. Costa, M. Ribeiro, F. Palomba, and I. Machado, "Automating test-specific refactoring mining: A mixed-method investigation," in *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023, pp. 13–24.

[25] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. L. M. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date?" *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1152–1170, 2023.