Journal of Software Engineering Research and Development, 2026, 14:3, doi: 10.5753/jserd.2026.5688

Refactoring Code Smells in Open Source Projects: A Hands-on Approach to Teaching Software Maintenance

Carla Bezerra [**Federal University of Ceara** | ***carlailane@ufc.br***]
Victor Anthony Alves [**Federal University of Ceara** | ***anthonyvictor90@gmail.com***]
Antônio Hugo Lobo [**Federal University of Ceara** | ***hugorplobo@alu.ufc.br***]
João Paulo Queiroz [**Federal University of Ceara** | ***Joaop3595@gmail.com***]
Lara Lima [**Federal University of Ceara** | ***laragabriellysouzabatista@gmail.com***]
Paulo Meirelles [**University of São Paulo** | ***paulormm@ime.usp.br***]

Abstract Code smells are suboptimal structures that undermine software quality. While refactoring is the standard technique to address them, its manual application can degrade code if done without discipline. Despite its importance, refactoring is rarely explored in depth in undergraduate computing courses, creating a gap between academia and industry. Simultaneously, Open Source Software (OSS) projects offer authentic, hands-on learning environments for software maintenance. To address the academic gap and leverage this opportunity, this paper presents and evaluates a hands-on pedagogical approach for teaching code smell refactoring through student contributions to OSS projects. We implemented this approach in two undergraduate Software Quality and Maintenance courses. Our analysis of students' learning experiences reveals that they recognized quality improvements and the connection between refactoring and testing. However, they faced challenges with code complexity and cross-file changes, which sometimes inadvertently introduced new code smells. Regarding the OSS experience, students reported professional growth but struggled with contribution workflows and receiving feedback from maintainers. Our findings offer valuable insights and propose actionable pedagogical recommendations for educators seeking to integrate advanced software maintenance practices into their curricula by leveraging the real-world environment of OSS.

Keywords: *Code Smells, Refactoring, Software Engineering Education, Open-source Software*

1 Introduction

Code smells can indicate problems related to aspects of code quality, such as readability and modifiability (Fowler, 2018). These anomalies can affect any software system (Oizumi et al., 2016). Unlike bugs or defects, the presence of code smells does not necessarily imply functional errors. However, they may have other negative consequences, particularly impacting software maintenance and evolution (Lacerda et al., 2020).

Refactoring can be used to remove code smells and directly improve code quality (Yamashita and Moonen, 2013). Fowler (2018) defines refactoring as a series of small changes to a codebase's internal structure that do not alter its external behavior. In practice, refactoring serves multiple purposes, such as mitigating software design degradation, reducing maintenance effort, facilitating the implementation of new features, fixing existing bugs, and eliminating code smells (Mariani and Vergilio, 2017; Golubev et al., 2021; Halepmollasi and Tosun, 2024).

Social coding and Open Source Software (OSS) have gained significant traction, offering developers diverse opportunities for community engagement and collaborative work (Jiang et al., 2017). This paradigm enables external contributors to propose changes to a software project without needing direct access to its central repository (Yu et al., 2016). High-quality OSS projects often depend on large and sustainable communities to develop features, fix bugs, and maintain code quality (Aberdour, 2007). Recent studies indicate, however, that OSS development involves risks related to software quality and long-term maintenance (Berg et al., 2022; Haider et al., 2023; Jin et al., 2023). One significant risk factor concerns the maturity of development processes, especially the quality of code submitted by contributors (Qiu et al., 2023). Identifying competent contributors in a vast pool of candidates is often challenging and time-consuming for OSS maintainers (Jin et al., 2023).

In educational contexts, OSS projects are frequently adopted by software engineering instructors as part of assessment activities, offering students valuable experiential learning opportunities (Pinto et al., 2019). The involvement of students in OSS collaboration has been the focus of various studies (Morgan and Jensen, 2014; Nascimento et al., 2013; Diniz et al., 2017), which highlight that student contributions can have a positive impact on these projects. The most emphasized skills are code refactoring and issue resolution (Bezerra et al., 2022). Nonetheless, despite demonstrating these capabilities, many students lack the motivation to engage actively in improving real-world projects (Silva et al., 2020). Thus, encouraging student contributions to OSS is essential, as it allows them to apply and refine their skills while strengthening project quality (Pinto et al., 2019). This approach benefits OSS initiatives and provides students with meaningful, hands-on learning experiences that prepare them for careers in the software industry (Hu et al., 2019; Diniz et al., 2017; Ellis et al., 2010).

However, despite the recognized importance of these skills, refactoring and code smells are rarely explored in depth in undergraduate computing courses. This creates a significant gap between academic training and industry practice, where engineers must regularly address legacy code and design degradation.

In our previous study (Bezerra et al., 2024), we explored students' perceptions of refactoring practices, the most challenging code smells to address, the difficulties encountered during the process, the most frequently used refactoring techniques, the impact of refactoring on internal code quality, the benefits of collaboration in OSS, and students' impressions of contributing to real-world projects.

This paper extends that investigation by presenting and evaluating this hands-on pedagogical approach to teaching refactoring through contributions to OSS. We conducted an experiment in two Software Quality and Maintenance classes, analyzing the perceptions of 29 students (who refactored 20 code smells across 23 OSS projects). The main objective is twofold: (i) to evaluate a pedagogical approach that integrates code smell refactoring into undergraduate education through real-world OSS contributions; and (ii) to empirically analyze how students apply these refactoring techniques, investigating their motivations, challenges, misconceptions, and whether their efforts lead to measurable improvements. This study includes new analyses from students' learning diaries (e.g., introduction of new smells) and proposes pedagogical recommendations derived from these findings.

Our contributions include (i) an empirical characterization of how students refactor code smells while contributing to OSS in an authentic educational setting, (ii) the identification of recurring patterns and difficulties in the refactoring process, and (iii) the formulation of pedagogical recommendations based on empirical evidence to assist instructors in planning and executing activities that promote software quality and develop critical technical skills.

The remainder of this paper is organized as follows. Section 2 presents theoretical foundations related to code refactoring and its relevance to Software Engineering education. Section 3 reviews related work that informs this study. Section 4 describes the methodological design, including course context, participant profile, and data collection and analysis procedures. Section 5 reports the results, focusing on refactoring types, underlying motivations, and challenges faced by students. Section 6 contextualizes the findings in light of the literature and discusses their educational implications. Section 7 addresses limitations and threats to validity. Finally, Section 8 concludes the paper and outlines future research directions.

2 Background

Code smells and refactoring practices, along with the characteristics of Open Source Software (OSS) development, form the foundation of our educational approach, in which students identify and resolve code smells in OSS projects through refactoring activities.

2.1 Code Smells and Refactoring

Code smells may indicate design problems at multiple granularity levels, such as method-level or class-level issues. Software developers often rely on code smells as indicators of poor code quality (Sousa et al., 2018; Fowler, 2018), and actively seek to identify code smells and anti-patterns in their source code (Tahir et al., 2018). These anomalies do not necessarily correspond to functional defects but can hinder software evolution and maintainability.

Table 1. Code smells detected by PMD

Code Smells	Description
Excessive Method Length	A method with a high number of lines of codes (Khomh et al., 2009)
Excessive Class Length	A class with a high number of lines of codes (Khomh et al., 2009)
Double Checked Locking	An object is initialized but not all object fields are necessarily written to the heap (Farchi et al., 2003).
Duplicated Code	Identical or very similar pieces of code (Fowler, 2018)
God Class	Too many software features in a class. It tends to be very large and hard to read and understand (Fowler, 2018)
Data Class	Classes that have fields, getting and setting methods for the fields, and nothing else (Fowler, 2018)
Collapsible If Statements	A series of nested 'if' statements anderson2020addressing
Excessive Parameter List	Function or method that have too many parameters (Fowler, 2018)
Simplified Ternary	Excessive or inappropriate use of the ternary operator
For Loop Variable Count	When the control variable of a for loop is used for purposes other than controlling the number of iterations of the loop

Table 2. Code smells detected by ReactSniffer

Code Smells	Description
Any Type	The use of the 'any' type in TypeScript disables the of types, compromises security and can lead to errors at runtime
Many Non-Null Assertions	The non-null assertion operator ('!') in TypeScript ignores type checks and can cause errors at runtime
Missing Union Type Abstraction	Type aliases in TypeScript allow you to define reusable types and accept union types, making code easier to maintain, read, and readability of the code
Enum Implicit Values	When enums are used without explicit values defined
Too Many Props	When a component receives and uses many props (Ferreira and Valente, 2023)
Large File	Files with a lot of lines and components (Ferreira and Valente, 2023)
Large Component	Components that are difficult to read because they have a large number of lines of code (Ferreira and Valente, 2023)
jSX Outside the Render Method	When JSX code is outside the render method of the component may indicate that the component has too much responsibility (Ferreira and Valente, 2023)
Force Update	When a piece of code forces a component or page to reload (Ferreira and Valente, 2023)
Uncontrolled Component	A component that does not use props/state to handle form's data (Ferreira and Valente, 2023)

There are several tools available for detecting code smells (Fernandes et al., 2016). In this study, we used two tools to support our analysis: `PMD`[1], which detects code smells in Java projects, and an extension of the `React Sniffer`[2] tool, designed to detect code smells in React projects that use the `TypeScript` language[3]. These tools have been adopted in previous research (Ferreira and Valente, 2023; Fernandes et al., 2020; Martins et al., 2020, 2021) and provided a foundation for students to identify and address design problems in real-world software systems.Tables 1 and 2 present the sets of code smells considered in this study for Java and React

[1] `https://pmd.github.io/`
[2] `https://github.com/fabiosferreira/reactsniffer`
[3] `https://github.com/maykongsn/reactsniffer`

projects, respectively.

These tools enabled students to detect design anomalies in the codebases of OSS projects and apply refactoring techniques to mitigate them. This practical experience helped them reinforce their understanding of core software quality principles.

2.2 Open Source Software

Open Source Software (OSS) is developed through large-scale collaboration within digital communities, offering users free and accessible alternatives to proprietary software (Dong and Götz, 2021; Hoffmann et al., 2024). OSS projects are characterized by voluntary participation and distributed contributions, often relying on peer review, community feedback, and transparent development practices (Aberdour, 2007).

This development model creates opportunities and challenges. On the one hand, it fosters innovation, code reuse, and knowledge sharing. On the other hand, the variability in contributors' experience levels and the lack of centralized control can make it difficult to maintain high-quality codebases (Berg et al., 2022; Haider et al., 2023; Jin et al., 2023). As a result, OSS projects are susceptible to the accumulation of technical debt, including code smells (Nanthaamornphong and Boonchieng, 2023; Qiu et al., 2023).

Several studies have examined the occurrence of code smells in OSS contexts. For instance, Nanthaamornphong and Boonchieng (2023) analyzed code smells during the code review process, Kaur et al. (2021) surveyed detection techniques, and Nanthaamornphong et al. (2020) developed tools specific to OSS environments. These efforts highlight that managing code smells remains a persistent challenge and a relevant topic for software engineering research.

Moreover, OSS provides a promising educational setting. Instructors can leverage real-world projects to expose students to authentic development practices and foster skills such as testing, debugging, code review, and refactoring (Pinto et al., 2019; Nascimento et al., 2013; Diniz et al., 2017). With proper guidance, students can contribute to the enhancement of established codebases, thereby acquiring practical experience that aligns with industry requirements.

3 Related Work

Code smells in OSS projects are a well-documented research area. Studies have investigated their presence at various development stages, such as during code review (Nanthaamornphong and Boonchieng, 2023), evaluated the efficacy of different detection techniques (Kaur et al., 2021), and proposed specific tools for their identification in OSS environments (Nanthaamornphong et al., 2020). This body of work highlights that the recurrence of code smells is a persistent challenge and a relevant topic of investigation in OSS.

In parallel with research on OSS practices, several studies have investigated educational strategies for teaching code quality and refactoring. For instance, Keuning et al. (2019, 2021) explored how code quality is addressed in software engineering education. In 2019, they investigated teachers' practices, identifying challenges and proposing guidelines for teaching code quality. These guidelines, grounded in teacher cognizance and code quality tools, informed the evaluation criteria in our study. Later, Keuning et al. (2021) proposed a tutoring system to support students in improving code while preserving functionality. This system inspired aspects of our instructional approach to teaching refactoring practices.

Beyond general strategies and tutoring systems, gamification has also been explored as a pedagogical tool that engages learners. Refactor4Green (Agrahari and Chimalakonda, 2020) is a gamified educational tool that teaches refactoring with a focus on energy efficiency, using learning cards and quizzes to reinforce key concepts. Similarly, dos Santos et al. (2019) introduced CleanGame, a gamified platform for identifying code smells in Java programs. Both studies illustrate how gamification can effectively engage students in learning refactoring. In contrast, our approach emphasizes authentic software engineering tasks, as students contribute directly to OSS projects and experience real-world constraints and collaborative dynamics.

This focus on authentic OSS contributions builds on work such as that of Pinto et al. (2019), who examined students' perceptions of contributing to OSS projects and emphasized the importance of teacher facilitation. Their results show that students view this experience positively. Our study builds on this by incorporating OSS contributions specifically designed to identify and remove code smells. We extend our prior work by analyzing the technical and pedagogical dimensions of these contributions, including unintended outcomes such as the introduction of new smells.

While our study focuses on manual refactoring, other research has analyzed the use of automated tools. AlOmar et al. (2024) analyzed how students apply refactoring using automated tools, focusing on usability and educational effectiveness. Their study combined code quality metrics with a qualitative survey, finding that tool-assisted refactoring can improve code quality. Our work shares a similar analysis of refactoring outcomes but differs by focusing on manual, student-driven refactoring within OSS contributions, which adds complexity and realism to the learning experience.

In a similar vein of exploring modern tools, Menolli et al. (2024) proposed a refactoring teaching strategy that integrates Generative AI (GAI), such as ChatGPT, into a learning model based on single- and double-loop reflection. Their study showed that GAI can support the development of computational thinking skills, including problem formulation and abstraction. While our work shares this emphasis on hands-on learning, our approach centers on collaboration and contribution to real-world projects, offering insights into the challenges of applying refactoring practices in authentic development environments.

4 Study Settings

This study has two complementary objectives: (i) evaluate a pedagogical approach that integrates code smell refactoring into undergraduate education through contributions to `Java` and `React` open-source projects, and (ii) investigate whether students' refactoring efforts lead to measurable improvements in internal quality attributes, providing concrete

value to OSS projects. By combining an educational perspective with a technical evaluation, the study provides insights into how students learn and apply refactoring techniques while examining the tangible effects of their contributions on software quality in real-world open-source projects.

4.1 Research Questions

To guide our evaluation of this hands-on pedagogical approach and to empirically analyze its implementation (as stated in our main objective), we have defined the following research questions (**RQs**):

RQ$_1$ *– What is students' perception about the practice of code smell refactoring?* **RQ**$_1$ seeks to understand students' perception of the importance and value of code smell refactoring practices. By answering RQ$_1$, we can gain insight into students' opinions on employing refactoring techniques and identify which refactoring approaches are used most frequently. While the literature suggests specific refactoring techniques for some code smells, the choice of refactoring techniques is flexible for most code smells. We aim to determine whether students are applying refactoring techniques effectively.

RQ$_2$ *– What code smells are harder to refactor according to students' perceptions?* **RQ**$_2$ aims to identify the types of code smells students perceive as most challenging to refactor. To answer **RQ**$_2$, we quantitatively analyze which code smells were identified by students as the most difficult to address and examine the refactoring techniques employed for each type of smell.

RQ$_3$ *– Was there any refactoring that added code smells?* By answering **RQ**$_3$, we can determine whether code smells were introduced after refactoring and, if so, identify which ones were added most frequently. Additionally, we analyze the relationship between refactoring duration and the likelihood of introducing new code smells.

RQ$_4$ *– What are the difficulties of refactoring code smells according to students' perceptions?* By answering **RQ**$_4$, we can identify students' most common challenges during refactoring practices. Additionally, we can analyze the relationship between encountered difficulties and refactoring outcomes, including their impact on internal quality attributes.

RQ$_5$ *– How was the process of students contributing to OSS projects?* **RQ**$_5$ aims to explore the contribution process that students undergo when working on OSS projects. This analysis examines the challenges they encounter, the steps they follow to submit contributions, their interactions with project maintainers, and their adaptation to different project workflows and requirements.

RQ$_6$ *– What are students' perceptions of the benefits of collaboration in OSS projects from the perspective of improving code quality?* **RQ**$_6$ aims to identify the benefits of improving code quality from the students' perspective after they submit refactoring contributions to OSS projects. By answering **RQ**$_6$, we can understand how students perceive the impact of their refactoring efforts on overall code quality and the collaborative aspects of working within OSS communities.

RQ$_7$*: What are students' perceptions of the contribution of refactoring in OSS projects?* **RQ**$_7$ investigates students' perceptions of the refactoring contribution process in OSS projects. By answering **RQ**$_7$, we can understand how students experience the process of submitting contributions according to project standards, including their interactions with maintainers and responses to contribution acceptance or rejection.

4.2 Steps and Procedures

We conducted the following steps to perform code smell refactoring practices with the students:

Step 1: Theoretical and Practical Content Training. In the initial phase, we provided theoretical instruction in the Software Quality and Software Maintenance courses, covering essential concepts for refactoring code smells. This phase consisted of three weeks of theoretical classes, totaling 12 hours, which focused on refactoring techniques and identifying code smells. Following this, two weeks, comprising 8 hours, were dedicated to exploring metrics for internal quality attributes (complexity, size, cohesion, coupling, and inheritance), which will be discussed in more detail later in the study, along with practical sessions utilizing the `Understand tool`[4]. Additionally, students participated in a 2-hour workshop session discussing contributions to open-source software. Concluding the curriculum component, one week, involving 4 hours, was allocated to instructing students on effectively utilizing the `PMD` and `ReactSniffer` tools. This comprehensive educational program spans 26 hours, representing a pivotal enhancement to the course's academic offerings.

The classes focused on Software Quality (**C1**) and Software Maintenance (**C2**) were held during the academic period of 2023.2, primarily involving students from the undergraduate Software Engineering program. Table 3 profiles students engaged in the practice of code smells refactoring, detailing their participation across several dimensions: **ID** identifies the student; **Semester** specifies the academic term during which the student was enrolled in the course; **Programming Language** indicates proficiency in the selected project language (Java or React TypeScript); **Code Smells** shows whether the student already knew code smells before the course; **Refactoring** represents the student's level of knowledge in refactoring techniques; **Open Source** indicates whether the student had previously contributed to open source projects.

Step 2: Presentation of the code smells refactoring practice. After introducing the concepts and tools to the students, we presented the code smell refactoring practice tasks (Tn) that participants should carry out, which constituted the final work of the course. Working in teams of two people or individually, students were required to select an OSS project in Java or React from a provided list to identify and refactor the project's code smells. Table 4 shows the division of students by class. Each team had to refactor at least four different types of code smells, with a minimum of 20 occurrences

[4] `https://scitools.com/`

Table 3. Students profile

ID	Semester	Programming Language	Code Smells	Refactoring	Open Source
P1	8th	Minimum	No	None	No
P2	6th	Intermediate	Yes	Basic	No
P3	6th	Advanced	Yes	Intermediate	No
P4	8th	Advanced	Yes	Intermediate	No
P5	6th	Advanced	No	Intermediate	No
P6	4th	Intermediate	No	Intermediate	No
P7	6th	Advanced	No	Intermediate	No
P8	8th	Intermediate	Yes	Basic	No
P9	6th	Advanced	Yes	Intermediate	No
P10	6th	Minimum	No	Minimum	No
P11	8th	Intermediate	No	Minimum	No
P12	6th	Minimum	Yes	Minimum	No
P13	8th	Intermediate	Yes	Intermediate	No
P14	8th	Basic	Yes	Advanced	No
P15	6th	Basic	Yes	Intermediate	No
P16	6th	Intermediate	Yes	Basic	No
P17	8th	Intermediate	Yes	Basic	No
P18	6th	None	No	Minimum	No
P19	8th	Advanced	Yes	Intermediate	No
P20	8th	Intermediate	No	Basic	Yes
P21	4th	Intermediate	No	Minimum	No
P22	7th	Minimum	No	Basic	No
P23	10th	Advanced	No	Basic	No
P24	6th	Basic	Yes	Advanced	No
P25	10th	Intermediate	Yes	Intermediate	No
P26	6th	Basic	Yes	Basic	No
P27	6th	Intermediate	No	Minimum	Yes
P28	7th	Intermediate	Yes	Intermediate	No
P29	8th	Intermediate	No	Basic	No

of code smells across these four types.

Table 4. Students divided by class

Class	Students	Total
Class 1 (C1)	P1, P2, P3, P4, P6, P7, P9, P10, P11, P12, P16, P18, P21, P22, P23, P25, P26, P27, P28, P29	20
Class 2 (C2)	P8, P13, P14, P17, P19, P20	6
Both classes	P5, P15, P24	3

The practice delivery was divided into the following tasks:

T1 - Choose a project and contribute to OSS projects. For project selection, students adhered to the following criteria: (i) projects must be implemented in Java or TypeScript, (ii) contain a minimum of 2,000 lines of code, and (iii) exhibit at least four types of code smells with a total of 20 occurrences or more. In cases where a project did not meet the specified criteria, students were required to select an additional project to fulfill the quota. Students were also encouraged to solve minor issues in these OSS projects. Table 5 provides an overview of the selected projects, including project ID, team composition, students involved in refactoring, programming language, and total lines of code.

T2 - Identify and address code smells. Students were allowed to choose four types of code smells freely if the analysis tool detected more than four types of code smells. Using `PMD` or `ReactSniffer`, they identified and addressed 20 occurrences of these code smells. This approach enabled students to focus on the most prevalent or relevant areas of code quality, thereby enhancing their understanding of refactoring techniques and software design principles. Table 6 illustrates the number of refactored code smells documented in students' diaries and the corresponding programming languages.

T3 - Measurement of Code Quality. Students employed the `Understand tool` to assess code quality before and after refactoring, aiming to quantify internal quality attributes as outlined in Table 7: Cyclomatic Complexity (CC), Sum Cyclomatic Complexity (SCC) Average Cyclomatic Complexity (ACC), Nesting (MaxNest), CountDeclFunction, CountLine, Comment Lines of Code (CLOC), Lack of Cohesion of Methods (LCOM2), Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Number Of Children (NOC), and Base Classes (IFANIN). By analyzing these metrics, students can observe improvements in various aspects of code quality, thereby gaining a deeper understanding of the effectiveness of the refactoring process.

Table 5. Project characterization

System	Teams	Students	Language	LOC
Marine-api	T1	P27	Java	43145
Gitlab4j-api	T2	P6, P21	Java	73542
Scholarx-backend-v1	T3	P11	Java	5571
AppointmentScheduler	T4	P29	Java	6129
shopify-sdk	T5	P25, P7	Java	14313
yahoofinance-api	T6	P26, P2	Java	5126
repodriller	T7	P17	Java	7560
smart-doc	T8	P13, P8	Java	16989
ck	T9	P14	Java	96523
spotify-web-api-java	T10	P28	Java	114600
slim	T11	P1	React	17490
genshin-music	T12	P4	React	58937
neodash	T13	P9, P3	React	24317
react-design-editor	T14	P12	React	32058
github-profilinator.git github-profilinator.git github-profilinator.git	T15	P22	React	3870, 4893, 134996
open-tacos	T16	P15, P24	React	28549
WoWAnalyzer	T17	P10, P16	React	334060
editor	T18	P18	React	29723
reaction	T19	P19	React	51042
twilio-video-app-react	T20	P20	React	104717
saleor-dashboard	T21	P5	React	264719

Table 6. Frequency of Code Smells refactored

Code Smells	Frequency	Language
Any Type	63	React
Missing Union Type Abstraction	53	React
Excessive Method Length	37	Java
Large Component	35	React
Too Many Props	28	React
Excessive Class Length	28	Java
JSX Outside The Render Method	20	React
Many Non-Null Assertions	14	React
Double Checked Locking	12	Java
Enum Implicit Values	10	React
Large File	9	React
Duplicated Code	6	Java
Data Class	6	Java
God Class	5	Java
Collapsible If Statements	3	Java
Excessive Parameter List	3	Java
Force Update	3	React
Simplified Ternary	2	Java
For Loop Variable Count	1	Java
Uncontrolled Component	1	React

T4 - Refactor one type of code smell at a time, documenting challenges and techniques. Following the initial analysis using the `Understand tool`, students proceeded to refactor the identified code smells systematically, addressing one type at a time. After completing the refactoring process of each kind, students were required to use the diary technique (França et al., 2020). In this diary, they documented the difficulties encountered, the refactoring methods applied, and any observed changes, including the potential introduction of new code smells or improvements in quality metrics. Table 8 presents the diary structure, from which the responses were analyzed to assess whether refactoring could lead to the emergence of additional code smells. Additionally, the diary recorded the time each student spent refactoring each code smell.

Table 7. Internal quality metrics analyzed

Quality attributes	Metrics	Description
Complexity	McCabe Cyclomatic Complexity (CC)	It is equal to the number of decision points contained in that program plus one. The higher the value of this metric, the more complex the code structure (McCabe, 1976).
	Sum Cyclomatic Complexity (SCC)	Sum of cyclomatic complexity of all nested functions or methods. The higher the value of this metric, the more complex the code structure (McCabe, 1976).
	Average Cyclomatic Complexity (ACC)	Average cyclomatic complexity for all nested functions or methods. The higher the value of this metric, the more complex the code structure (McCabe, 1976).
	Nesting (MaxNest)	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function. The higher the value of this metric, the more complex the code structure (Lorenz and Kidd, 1994).
Size	CountDeclFunction	Number of functions. The higher the value of this metric, the larger the system size (Inc., 2023).
	CountLine	Number of physical lines. The higher the value of this metric, the larger the system size (Inc., 2023).
	Comment Lines of Code (CLOC)	Number of lines containing a comment. The higher the value of this metric, the larger the system size (Lorenz and Kidd, 1994).
Cohesion	Lack of Cohesion of Methods (LCOM2)	Calculates what percentage of class methods use a given class instance variable. The higher the value of this metric, the less cohesive the class (Chidamber and Kemerer, 1994).
Coupling	Coupling Between Objects (CBO)	Number of classes that a class is coupled. The higher the value of this metric, the more coupling there is between classes and methods (Chidamber and Kemerer, 1994).
Inheritance	Depth of Inheritance Tree (DIT)	Maximum depth of class in inheritance tree. The higher the value of this metric, the greater is the degree of inheritance of a system (Chidamber and Kemerer, 1994).
	Number Of Children (NOC)	Number of immediate subclasses. The higher the value of this metric, the greater the degree of inheritance of a system (Chidamber and Kemerer, 1994).
	Bases Classes (IFANIN)	Number of immediate base classes. The higher the value of this metric, the greater the degree of inheritance of a system (Destefanis et al., 2014).

Table 8. Diary structure

Questions
I'm currently working on refactoring the following code smell:
My main difficulties in removing these anomalies are:
I am using the following refactoring methods to remove code smells:
When refactoring this smell, I realized whether or not the following code smells were inserted:
When refactoring this smell, I noticed whether or not there was an improvement in the following quality metrics:
Report the average amount of time it took to refactor the code smell:

T5 - Conduct a code review and submit contributions, specifying improvements made based on metrics. During the final phase, students conducted code reviews of their refactored code and submitted contributions to the project repository. Based on the gathered metrics, they identified improvements, addressed code quality enhancements, and resolved any remaining code smells. In addition to these quantitative aspects, we qualitatively analyzed students' diaries and repository activity to gain a deeper understanding of how the contribution process unfolded from multiple perspectives.

After completing the code smells refactoring task, students were asked to answer a questionnaire to evaluate their experience. The questionnaire covered various aspects, including the challenges faced when refactoring the most complex code smells, the difficulties encountered during the refactoring process and collaboration in OSS projects, and the perceived benefits of these practices. We invited students to share their perceptions about the most challenging aspects of refactoring, their specific difficulties, and the benefits they identified from collaborating on OSS projects.

4.3 Qualitative analysis

With these individual perceptions as a basis, we conducted a qualitative analysis to explore how students experienced the applied refactoring practice. This analysis allowed us to identify recurring themes in their reflections, understand their emotional responses to engaging with a real codebase, and examine the development of hard skills, such as debugging, testing, and the application of refactoring techniques, within the context of open-source contributions.

All qualitative analyses in this study were conducted using thematic analysis, following the six-phase process proposed by Braun and Clarke (2006): (1) data familiarization, (2) initial code generation, (3) theme identification, (4) theme review, (5) theme definition and naming, and (6) report writing. To ensure the reliability and validity of the coding process, three researchers worked independently to analyze the data.

Each researcher conducted their coding separately, without knowledge of the others' coding decisions. This independent coding allowed for a more objective and unbiased assessment of the data. Subsequently, the researchers compared their coding results. In instances where discrepancies or disagreements arose between the researchers' coding, they engaged in a thorough discussion to resolve them. This collaborative discussion aimed to explore the reasons behind the differing interpretations and to consider the nuances of the data carefully. Through this process of discussion and deliberation, the researchers worked towards reaching a consensus on the appropriate coding for each data point. This consensus-building approach was implemented to guarantee the systematic and rigorous identification of recurring patterns and themes evident across the students' responses, thereby strengthening the credibility of the findings.

4.4 Questionnaires structure

The flow of tasks is illustrated in Figure 1. We used two primary instruments to evaluate the students' learning experience and perceptions: a pre-questionnaire and a post-questionnaire.

The **pre-questionnaire** (*Prior Knowledge Level about Code Smells and Collaboration in OSS Projects*- file available in our replication package) was administered immediately after the conclusion of the theoretical and practical content training (Step 1) and before students commenced the refactoring practice (Step 2). Its main objective was to establish a baseline of the participants' prior knowledge and experience following the introductory classes. Specifically, it captured students' self-assessed familiarity (using a scale from None to Expert) with concepts such as Refactoring, Code Smells (including types known and refactored), and proficiency in the programming languages used (`Java` and `React TypeScript`). It also assessed their previous experience with OSS collaboration. The instrument was structured with several questions, predominantly employing multiple-choice items and 5-point Likert scale items to gauge concep-

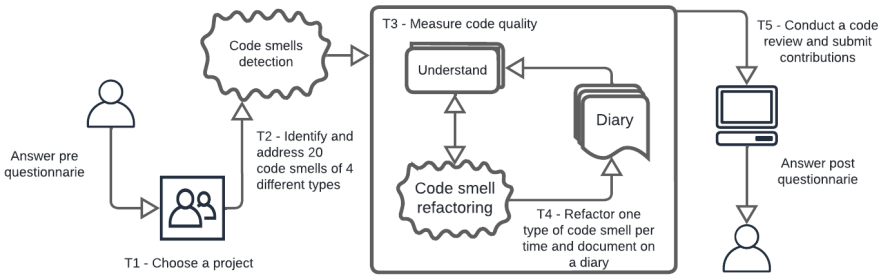

Figure 1. Tasks from practice delivery

tual perception.

The **post-questionnaire** (*Form for the knowledge level after the refactoring work on Code Smells and Collaboration in OSS projects* - file available in our replication package) was applied immediately following the completion of the entire refactoring and contribution practice (T5). Its purpose was threefold: 1) To measure the perceived change in knowledge regarding Code Smells and Refactoring; 2) To identify the Code Smells students found most challenging to refactor and document the primary difficulties encountered during the process; and 3) To evaluate the practical experience with OSS, covering logistical aspects (for example, project activity, number of issues submitted/accepted), the perceived difficulty of collaboration, and students' perception of the benefits of OSS collaboration in terms of improving code quality. This instrument consisted of 21 questions, including a combination of quantitative scales and several open-ended questions designed to facilitate the subsequent qualitative analysis. The questionnaires, answer spreadsheets, and all related materials are available in the replication package[5].

With these individual perceptions as a basis, we were able to explore how students experienced the applied refactoring practice.

5 Results

In the following sections, we present our findings and discuss the results, explicitly addressing the research questions posed in this study.

5.1 Students' perception of refactoring practices

We addressed $\mathbf{RQ}_1$ by analyzing the data collected through a questionnaire answered by the students. We asked them about the main benefits they identified in the practice of refactoring code smells (See Table 9) and what soft and hard skills they acquired through the execution of the work (Tables 10 and 11).

Table 9 shows that the majority of students considered that the practice of refactoring code smells led to an improvement in the quality of the systems. In addition, other benefits mentioned were improving their skills in identifying code smells, understanding a project, performing code cleaning, and preventing future problems. This suggests that refactoring code smells tends to have a positive impact on the quality of a system, according to the students' perceptions. Some student reports corroborate this statement:

[5] https://doi.org/10.5281/zenodo.13010596

Table 9. Students' perceptions about the benefits of code smells refactoring

Benefits	Students	Total
Improve software quality	P2, P5, P6, P7, P8, P10, P13, P15, P17, P20, P21, P22, P23, P24, P26, P27, P28, P29	18
Identification of code smells	P1, P9, P11, P25	4
Understand the project	P4, P6, P11, P14	4
Code cleanup	P3, P26, P28	3
Improve programming skills	P12, P16, P18	3
Removal of future problems	P3	1
Code review	P19	1

P2:"Making the code more maintainable and improving understanding of certain parts of the code."

P22:"Improving code readability and maintainability."

P26:"Code becomes more readable and clean."

Table 10. Students' perceptions about the soft skills acquired with the code smells refactoring

Soft Skills	Students	Total
None	P1, P5, P7, P9, P11, P12, P13, P14, P15, P16, P19, P20, P23	13
Creativity	P3, P18, P21, P26, P29	5
Problem-solving	P4, P6, P21, P26, P29	5
Teamwork	P8, P22, P24, P26	4
Critical thinking	P3, P6, P26	3
Adaptability	P3, P26, P29	3
Proactive	P2, P28	2
Communication	P19, P25	2

Table 10 indicates that several students (13) did not perceive any acquisition of soft skills after practicing code smell refactoring. It is essential to note that this does not mean practicing refactoring does not lead to the development of soft skills, as these students may consider themselves experienced in the soft skills employed during their work, such as creativity, problem-solving, and teamwork, which other students have mentioned.

Finally, Table 11 shows that, according to students' perceptions, the primary hard skills developed were debugging, testing, and refactoring techniques. This suggests an association between the use of tests and debugging tools as aids in refactoring, as proposed in the literature (Fowler, 2018), and knowledge of refactoring techniques. Some reports from students reinforce this statement:

P2:"Hard skills would be refactoring techniques, and in soft skills, it was being more proactive in problem-solving."

P21:"Hard skills: testing and debugging. Soft skills: problem-solving and creativity."

Table 11. Students' perceptions about the hard skills acquired with the code smells refactoring

Hard Skills	Students	Total
Debugging	P6, P9, P10, P16, P17, P20, P21, P26	8
Testing	P9, P10, P16, P17, P20, P21, P26, P27	8
Refactoring techniques	P2, P7, P12, P15, P28	5
None	P1, P3, P11, P18, P23	5
TypeScript	P4, P15, P22, P24	4
Algorithms	P10, P13, P27, P29	4
Java	P14, P25, P27	3
React	P15, P19, P24	3
Clean code	P4, P8	2
Code analysis	P5, P12	2
Git	P4	1
Github	P4	1
Maintenance	P27	1

Implications of $\mathbf{RQ}_1$. Our findings imply that most students reported that code-smell refactoring improved the system's quality. Furthermore, during the refactoring process, students perceived the acquisition of hard skills such as testing and debugging, which suggests a relationship between such skills and the refactoring process, as already proposed in the literature. However, several students also reported no acquisition from the soft skills standpoint.

5.2 Students' perceptions of the code smells that are harder to refactor

We analyzed the student responses after refactoring code smells to answer the $\mathbf{RQ}_2$. We conducted a new qualitative analysis to identify which code smells were the most difficult to remove. From the eight refactored code smells in the React projects, *Large Component* and *Too Many Props* are particularly noteworthy. In Java projects, out of the 15 refactored code smells, the code smells *God Class, Data Class*, and *Large Class* were the most frequently cited by students as the most difficult to refactor. Table 12 presents the most challenging code smells to refactor and the refactoring techniques students use.

Table 12. Code smells harder to refactor and the refactoring techniques used

Code Smells	Language	Students	Refactoring Techniques	Total
God Class	Java	P2, P6, P7, P3, P14, P25, P26, P28	Extract Class Extract Method Move Method	8
Large Component	React	P4, P5, P10, P15, P19, P20, P23, P24	Extract Component Extract Method Extract Class	8
Too Many Props	React	P5, P10, P12, P15, P18, P24, P25	Move Component Move Method Extract Interface	7
Large Class	Java	P14, P22, P25, P29	Extract Class Extract Method Move Method	4
Data Class	Java	P7, P25, P26, P29	Encapsulate Field	4

Table 12 reveals that the code smells described by students as the most difficult to refactor are related to highly robust and multifunctional classes (*God Class, Large Class, Data Class, Large Component, and Too Many Props*). Fowler (2018) points out that refactoring code smells requires techniques to divide the code into smaller, more manageable parts, each with a clear responsibility. In addition, he notes that refactoring these code smells can significantly impact various components of the code, revealing the need for a careful approach. These concepts may indicate that students have faced difficulties removing code smells that permeate multiple code files and often feel insecure about the impact of refactoring on responsibility-overloaded pieces of code. This observation suggests that refactoring these code smells requires a cautious and well-planned approach to ensure the effectiveness of the improvements made by the students.

However, students demonstrated a good understanding of refactoring practices and applied a multi-faceted approach to solving complex code problems. An example is the recurring use of techniques such as *Extract Component, Extract Method*, and *Extract Class* to address more complex code smells. These techniques aim to break down the code into smaller, more manageable parts, each with a clear responsibility (Fowler, 2018). This indicates that the students could select the most appropriate techniques for removing highly complex code smells, since all the code smells considered most difficult had the characteristic of a high burden of responsibility. It is clear, therefore, that despite the diversity of techniques available, many of them share similar goals and have been applied in a complementary way to achieve more effective results in improving project code quality.

Implications of $\mathbf{RQ}_2$. Our findings reinforce the assumption that students consider code problems that affect several source code files to be the most difficult to refactor. Students feel unconfident about removing these problems, mainly because they do not understand how other files are affected after refactoring. They are large and often complex files to refactor. In this way, we can infer that good programming practices and teaching focused on refactoring techniques facilitate the students' comprehension process. Based on these results, it is possible to compile a document that will serve as support to help students understand what each code smell is related to and the possible activities to address them. Additionally, it is interesting to observe a correlation between the most challenging code smells in React and Java. Students had more difficulty with code smells related to size and a high level of coupling, such as *God Class, Large Class, Large Component* and *Too Many Props*. However, they were able to choose the best refactoring techniques to reduce this coupling, such as *Extract Class* and *Extract Method.*

5.3 Refactoring leading to more code smells

From the collected diaries, we identified which code smells led to the introduction of other code smells ($\mathbf{RQ}_3$). Using the reported refactoring times, we analyzed the correlation between the time required to refactor specific code smells and those present in cases where new ones were introduced. Tables13 and 14 show the code smells that emerged in React and Java projects, respectively, following refactoring, based on an analysis of students' responses. The first column lists the refactored code smells, while the second indicates the newly introduced ones.

Table 13. Inserted code smells in React projects

Refactored code smell	Inserted code smell
Any Type	Any Type
JSX outside the render method	Any Type
Too Many Props	Any Type

Table 13 always indicates the *Any Type* smell whenever a code smell is introduced through refactoring in React projects. This pattern suggests that students may have resorted to *Any Type* as a last resort when they were unable to determine the appropriate type for a given situation.

Table 14. Inserted code smells in Java projects

Refactored code smell	Inserted code smell
ExcessiveClassLength	GuardLogStatement
GodClass	GuardLogStatement
GodClass	ExcessiveClassLength
Duplicated Code	Duplicated Code
ExcessiveClassLength	ExcessiveClassLength

Furthermore, examining Table 14, we observe that the code smells introduced after refactoring were *Guard Log Statement* (2), *Excessive Class Length* (2), and *Duplicated Code* (1).

The introduction of *GuardLogStatement* suggests that students may have added log statements to verify conditions before executing code, possibly as a debugging strategy.

Additionally, *ExcessiveClassLength* only appeared when *GodClass* and *ExcessiveClassLength* were being refactored. This matter implies that while students may have successfully refactored the class to follow the Single Responsibility Principle (SRP), they were unable to reduce its length sufficiently to prevent it from still being considered a code smell.

Finally, the occurrence of *Duplicated Code*, along with other code smells that reappeared after refactoring, indicates that students may have struggled to effectively eliminate these issues, potentially reintroducing them in the process.

Table 15. Code smells Average Refactoring Time in React projects

Code smell	ART	Students	Total
Large Component	117.36	P17, P16, P15, P15, P5, P15, P15 P15, P15, P15, P15, P15, P15, P15 P14, P12, P11, P20, P19	19
Any Type	74.55	P19, P18, P17, P16, P15, P15, P15, P15, P14, P11	10
Too Many Props	151	P18, P17, P16, P15, P14, P11, P19	7
Missing Union Type	192.4	P18, P17, P16, P14, P11, P20	6
JSX outside the render	20	P14, P11, P19	3
Large File	255	P17, P19	2
Many Non-Null	82.5	P18, P20	2
Enum implicit values	5.5	P20, P19	2
Force Update	5	P19	1

Additionally, based on the students' diary responses, we computed the Average Refactoring Time (ART), in minutes, for each distinct type of code smell addressed, rather than for each occurrence. Table 16 presents the ART alongside the total number of occurrences refactored for each code smell in the Java projects. As shown in the table, GodClass, Duplicated Code, and DoubleCheckedLocking exhibit the highest ART values, each exceeding 100 minutes, suggesting a higher level of complexity or effort required for their resolution.

Examining Tables 13, 14, 15, and 16, we can correlate the time spent refactoring each code smell with the introduction of new code smells as a consequence. Among the refactored smells, God Class (Java), Duplicated Code (Java), and Too Many Props (React) stand out the most, each requiring an average of over 100 minutes to refactor. Following them, Any Type averaged 74.55 minutes, and in React projects, it was the only code smell introduced after refactoring. However, it is essential to note that certain code smells, such as Large File, were refactored by only a few students. This limited sample size makes it challenging to determine whether the assigned ART is truly representative.

Table 16. Code smells Average Refactoring Time in Java projects

Code smell	ART	Students	Total
ExcessiveMethodLength	59.3	P7, P7, P7, P7, P7 P8, P6, P3, P3	9
DataClass	13.75	P3, P3, P3, P3, P3, P3, P3, P4	8
ExcessiveClassLength	57.28	P6, P5, P3, P3 P3, P3, P7	7
Duplicated Code	120	P3, P3, P9, P8, P4 P6	6
GodClass	156	P6, P5, P8, P7, P7	5
ExcessiveParameterList	60	P7, P6	2
DoubleCheckedLocking	120	P2, P2	2
CommentSize	20	P8	1
CollapsibleIfStatements	10	P2	1

Implications of $\mathbf{RQ}_3$: Our findings indicate a correlation between the time spent refactoring a code smell and introducing new code smells during the process. This analysis highlights the importance of caution, especially when addressing code smells that require more time, as extended refactoring increases the risk of unintended issues. Furthermore, examining the code smells with the highest Average Refactoring Time (ART) is crucial. In React projects, these include Large File, Missing Union Type, Too Many Props, and Large Component. The highest ART values were observed in Java projects for GodClass, DoubleCheckedLocking, and Duplicated Code.

5.4 Difficulties of refactoring code smells

From the answers collected, we identified four categories of difficulties that most occurred among the students ($\mathbf{RQ}_4$): (i) difficulty in understanding the source code; (ii) emergence of bugs after refactoring a code smell; (iii) difficulty choosing the refactoring techniques; and (iv) lack of knowledge about technologies. Table 17 presents the categories identified after analyzing the students' responses. The first column lists the categories identified; the second contains the students who experienced the respective difficulties during the refactoring of the code smell.

Table 17. Difficulties identified by students in the practice of refactoring

Categories	Students	Total
Difficulties understanding the source code	P3, P6, P7, P8, P9, P10, P11, P12, P14, P18, P19, P21, P23, P24, P26, P27	16
Emergence of bugs after refactoring a code smell	P5, P6, P16, P20, P21, P24, P26, P28, P29,	9
Difficulty choosing the refactoring technique	P2, P6, P10, P13, P15, P25	6
Lack of knowledge about technologies	P1, P17, P12, P22, P23	5

Table 17 reveals that nine students encountered difficulties when refactoring code smells, which led to the introduction of bugs into the project. This fact suggests that the process of refactoring code smells should be carried out with caution, applying techniques that preserve the behavior of features after refactoring. Some reports corroborate this statement, as shown below:

P5:"It was difficult to ensure that these changes did not cause side effects in other parts of the project."

P26:"It was hard not to impact the complexity and architecture of the application in problematic ways"

We also identified, through the information in Table 17, that some students are in more than one category of difficulty in code smell refactoring. As a result, we can see relationships between the categories of difficulties students encounter. The relationship we highlight is that the more difficult the source code is to understand, the more complicated it is to refactor the code smell. Some student reports reinforce such a statement:

P8:" I had difficulties understanding the code and the logic behind it, to refactor it effectively."

P9:"Learning about the project I am starting the refactoring on was complicated."

Lastly, we analyzed the impact of refactoring code smells on the following internal quality attributes: *complexity* and *size* for React projects (see Table 18) and *cohesion, coupling, complexity*, and *inheritance* for Java projects (see Table 19), their respective metrics, and the percentage change in each metric value before and after refactoring for each system. To analyze the impact of code smell refactoring on attributes with multiple metrics, we compared the sum of the pre- and post-refactoring metrics using data collected from the Understand tool. The symbol ↑ represents an increase in the metric value, the symbol ↓ represents a decrease in the metric value, and the symbol indicates that the attribute value has not changed after refactoring the code smells. It is essential to note that as the cohesion value increases, this attribute has improved due to the greater cohesion of a class or method, thereby enhancing the system's quality. Attributes such as coupling, inheritance, and complexity should have low values to indicate an improvement in the system's quality. The size attribute, in turn, can show improvement or deterioration in quality, depending on the context in which it is being evaluated.

For Java projects, we observed that cohesion was reduced in three projects, improved in only one project, and remained unchanged in another. Coupling had mixed results, while project size and inheritance remained the same. However, complexity was reduced in four projects and remained unchanged in one project, with significant reductions in two projects (S8 by 25.13% and S3 by 10.90%). These two projects, with the most considerable complexity reduction, involved refactoring code smells related to large code entities: Excessive Method Length and Excessive Class Length in S3, and Excessive Method Length and God Class in S8, suggesting a possible correlation between these code smells and system complexity.

As seen in Table 18, the overall impact of refactoring on React systems was minimal, with slight increases in complexity and size. One explanation for these results could be the students' selection of code smells. Smells like Any Type and Missing Union Type Abstraction represent changes that the metrics may not capture. Therefore, it is evident that further research is needed to investigate the impact of refactoring React code smells on software quality.

According to data from a systematic review (Al Dallal and Abdin, 2018), some studies in the literature have identified that code smell refactoring does not continually improve internal quality attributes, often having a negative impact, as occurred in our study. Similarly, it is noteworthy that the ineffectiveness of refactoring may result from students' difficulties during the refactoring process.

Implications of RQ$_4$: Our findings indicate that students still encountered various difficulties while refactoring code smells even after training. Based on their responses, we emphasize that refactoring code smells should be carried out cautiously, applying techniques that preserve the behavior of features after refactoring. Additionally, the more difficult the source code is to understand, the more complex it becomes to refactor the code smell. Moreover, the analysis of quality metrics before and after refactoring revealed mixed results. There were mixed outcomes in Java projects, including cases of cohesion degradation. In React projects, there were slight overall increases in complexity and size. This point corroborates the idea that refactoring code smells may yield no results or even lead to adverse outcomes, which, in this case, we can theorize is a product of the difficulties students encounter.

5.5 Process of contributing to OSS projects

After analyzing student contributions to OSS projects, we examined **RQ$_5$** through student responses and repository data. We conducted a qualitative analysis of the repository assigned to each team, including issues created, pull requests submitted, and interactions with the maintainers. For each team, we assessed various qualitative aspects of the contribution process using questions such as "Does the repository provide contribution guidelines?", "Does the repository resolve existing issues?", and "Does the repository provide contribution guidelines?". This analysis allowed us to understand how the students navigated the contribution process and the challenges they encountered. Table 20 presents all the aspects investigated, and Table 21 shows the results for each team.

The data reveals key aspects of the students' contribution process. First, while 15 out of 20 teams analyzed existing issues, only two (T7 and T18) successfully solved them, according to Q1 and Q2 results, as shown in Table 21. This data suggests that although most teams engaged with the repository's problems, implementing a solution proved to be a significant challenge. Possible reasons include difficulty understanding the codebase, debugging issues, or the complexity of the problem. These findings underscore the need for improved guidance and resources to assist students in bridging the gap between identifying issues and resolving them.

Another important observation is that only six teams found the contributing guidelines based on Q3 results. This point indicates that the guidelines were unclear or that students did not actively seek them. Since contributing guidelines often outline best practices and repository-specific requirements, teams unfamiliar with them might have struggled with submission standards. The low number of teams that accessed these guidelines suggests that making them more visible and accessible could improve the overall contribution expe-

Table 18. Impact of refactoring on internal quality attributes of React projects

System		Complexity					Size	
		AvgCyclomatic	SumCyclomatic	Cyclomatic	MaxNesting	CountDeclFunction	CountLine	CountLineComment
S13 before refactoring		1.93	9594	3272	14	1933	57043	3413
	Total:				12881,93			62389
S13 after refactoring		1.93	9595	3273	14	1932	57071	3413
	Total:				12883,93			62416
	Result:				**↑ 0.02%**			**↑ 0.04%**
S14 before refactoring		1.78	8145	2592	17	1310	65067	3773
	Total:				10755,78			70150
S14 after refactoring		1.41	8141	2598	17	1316	65059	3752
	Total:				10757,41			70127
	Result:				**↑ 0.02%**			**↓ 0.03%**
S15 before refactoring		29	1108	407	110	230	7786	70
	Total:				1654			8086
S15 after refactoring		33	1206	444	131	246	7843	70
	Total:				1814			8159
	Result:				**↑ 9.67%**			**↑ 0.90%**
S16 before refactoring		86	824	294	71	168	18175	2339
	Total:				1275			20682
S16 after refactoring		90	829	302	80	170	18143	2348
	Total:				1301			20661
	Result:				**↑ 2.04%**			**↓ 0.10%**
S17 before refactoring		2227	22252	8550	1526	6061	184253	3795
	Total:				34555			194109
S17 after refactoring		2226	22249	8547	1520	6060	184233	3797
	Total:				34542			194090
	Result:				**↓ 0.04%**			**↓ 0.01%**
S18 before refactoring		561	2572	69	242	1542	28549	2294
	Total:				3444			32385
S18 after refactoring		551	2566	67	242	1551	28611	2294
	Total:				3426			32456
	Result:				**↓ 0.52%**			**↑ 0.22%**
S19 before refactoring		6778	70011	23085	6691	14874	728126	31418
	Total:				106565			774418
S19 after refactoring		6870	69938	23101	6718	14892	727729	31385
	Total:				106627			774006
	Result:				**↑ 0.06%**			**↓ 0.05%**
S20 before refactoring		1.22	5234	1619	5	1220	29723	2722
	Total:				6859,22			33665
S20 after refactoring		1.25	5234	1634	5	1220	29970	2722
	Total:				6874,25			33912
	Result:				**↑ 0.22%**			**↑ 0.73%**
S21 before refactoring		295	1448	31	101	817	17500	523
	Total:				1875			18840
S21 after refactoring		301	1388	31	99	787	17266	523
	Total:				1819			18576
	Result:				**↓ 2.99%**			**↓ 1.40%**
S22 before refactoring		228	2204	84	54	1967	14338	312
	Total:				2570			16617
S22 after refactoring		232	2214	84	59	1966	14322	293
	Total:				2589			16581
	Result:				**↑ 0.74%**			**↑ 0.22%**
S23 before refactoring		0	0	236407	397	13576	875	6568
	Total:				236804			21019
S23 after refactoring		0	0	236420	397	13605	879	6590
	Total:				236817			21074
	Result:				**↑ 0.01%**			**↑ 0.26%**

Table 19. Impact of refactoring on internal quality attributes of Java projects

System		Cohesion	Coupling	Complexity			Inheritance	
		PercentLack OfCohesion	CountClass Coupled	SumCyclomatic	MaxNesting	CountClass Derived	CountClass Base	MaxInhe ritanceTree
S1 before refactoring		6447	1848	12645	1012	289	463	573
	Total:	**6447**	**1848**		**13657**		**752**	**573**
S1 after refactoring		6302	1834	12487	920	289	463	573
	Total:	**6302**	**1834**		**13407**		**752**	**573**
	Result:	**↓ 2.25%**	**↓ 0.76%**		**↓ 1.83%**		**– 0.00%**	**– 0.00%**
S2 before refactoring		21368	4605	34648	1457	205	732	1024
	Total:	**21368**	**4605**		**36105**		**937**	**1024**
S2 after refactoring		21368	4605	34648	1456	205	732	1024
	Total:	**21368**	**4605**		**36104**		**937**	**1024**
	Result:	**– 0.00%**	**– 0.00%**		**– 0.00%**		**– 0.00%**	**– 0.00%**
S3 before refactoring		1866	791	2664	88	12	81	109
	Total:	**1866**	**791**		**2752**		**93**	**109**
S3 after refactoring		1438	791	2364	88	12	81	109
	Total:	**1438**	**791**		**2452**		**93**	**109**
	Result:	**↓ 22.94%**	**– 0.00%**		**↓ 10.90%**		**– 0.00%**	**– 0.00%**
S8 before refactoring		5224	1590	12499	1048	34	192	185
	Total:	**5224**	**1590**		**13547**		**226**	**185**
S8 after refactoring		5258	1622	9317	826	34	192	185
	Total:	**5258**	**1622**		**10143**		**226**	**185**
	Result:	**↑ 0.65%**	**↑ 2.01%**		**↓ 25.13%**		**– 0.00%**	**– 0.00%**
S9 before refactoring		18312	5483	59112	3751	328	999	1132
	Total:	**18312**	**5483**		**62863**		**1327**	**1132**
S9 after refactoring		18219	5480	59089	3748	328	999	1132
	Total:	**18219**	**5480**		**62837**		**1327**	**1132**
	Result:	**↓ 0.51%**	**↓ 0.05%**		**↓ 0.04%**		**– 0.00%**	**– 0.00%**

Table 20. Aspects of the contribution process

ID	Aspects
Q1	Were existing issues analyzed?
Q2	Was an existing issue solved?
Q3	Were the contribution guidelines found?
Q4	Was the pull request accepted on the first submission?
Q5	Was the refactoring (pull request) eventually accepted?
Q6	Was there a review process for pull requests?
Q7	Were changes requested by a reviewer?
Q8	Were there automated checks for pull requests?

rience.

Following Q4 results, the data also shows that only four teams had their initial pull requests accepted. This result reinforces the idea that students faced challenges in submitting code that met the maintainers' expectations. Factors such as unfamiliarity with the repository's structure, lack of adherence to coding standards, or insufficient testing may have contributed to these low acceptance rates. Providing more precise documentation, examples of well-structured pull requests, and mentorship could help improve these numbers.

Refactoring was even less successful, with only two teams accepting their refactoring in Q5. This suggests that modifying existing code while preserving its functionality is particularly challenging for students. The complexity of refactoring, combined with a lack of automated feedback or experience with large-scale code modifications, contributed to this

Table 21. Results of the contribution process

Team	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
T1	No	No	No	No	No	No	No	No
T2	Yes	No	Yes	No	No	No	No	No
T3	Yes	No	No	Yes	No	No	No	No
T4	Yes	No	No	No	No	Yes	No	Yes
T5	No	No	No	No	No	Yes	Yes	Yes
T6	Yes	No	No	No	No	No	No	No
T7	Yes	Yes	No	No	N/A	N/A	N/A	N/A
T8	No	No	Yes	No	N/A	Yes	Yes	No
T9	Yes	No	No	No	N/A	N/A	N/A	N/A
T11	Yes	No	No	N/A	N/A	N/A	N/A	N/A
T12	Yes	No	No	No	N/A	N/A	N/A	N/A
T13	Yes	No	Yes	Yes	Yes	Yes	No	Yes
T14	Yes	No	Yes	Yes	No	No	No	No
T15	Yes	No	No	N/A	N/A	N/A	N/A	N/A
T16	Yes	No	Yes	No	No	No	No	Yes
T17	N/A	No	No	Yes	Yes	Yes	Yes	Yes
T18	Yes	Yes	No	No	No	No	Yes	No
T19	Yes	No	No	N/A	N/A	N/A	N/A	N/A
T20	Yes	No	Yes	No	No	Yes	No	Yes
Yes total:	**15**	**2**	**6**	**4**	**2**	**6**	**4**	**6**

outcome. These findings suggest that students would benefit from structured guidance on best practices for refactoring, including how to validate changes through testing.

The most striking finding relates to the review process in Q6 and Q7. While six teams underwent a structured review flow, only four received explicit change requests from reviewers. This inconsistency suggests that review and validation processes were not uniformly applied across teams. A well-defined review system, including human feedback and automated checks, is important for maintaining code quality and providing students with valuable learning opportunities. The lack of a structured review flow for many teams may have limited their ability to refine their contributions and understand common pitfalls in software development.

Regarding automated checks, only six teams encountered them in their pull request submissions, as indicated in Q8. This suggests that automated validation tools were not consistently applied or enforced. Automated checks help identify errors early, ensuring that contributions meet baseline quality standards before humans review them. Increasing the use of these tools could provide students with immediate feedback and reduce the likelihood of rejection.

Finally, the presence of N/A values in several responses suggests that certain teams either did not execute specific steps in the contribution process or did not have sufficient information to assess their progress. Some teams never reached the pull request stage, while others may have worked on aspects that did not require refactoring or review. These gaps highlight the need for a more structured approach to student contributions, ensuring that all teams participate in the key phases of the process. Standardizing review workflows and improving onboarding could lead to a more effective learning experience and higher-quality contributions.

Implications of RQ_5: The findings indicate several challenges in the students' contribution process, emphasizing improved guidance, more straightforward contribution guidelines, and a more structured review system. The difficulty in solving issues and the low acceptance rate of pull requests suggest that students struggle to understand existing codebases and adhere to repository standards. Additionally, the inconsistencies in review processes and the limited number of explicit change requests indicate that students may not always receive the necessary feedback to refine their contributions. The underuse of automated checks further suggests that early validation mechanisms were not effectively leveraged to assist students. To enhance the learning experience and improve the quality of contributions, it is crucial to establish a more standardized review process, encourage detailed feedback from reviewers, and integrate automated validation tools more consistently. Furthermore, making contribution guidelines accessible and ensuring all teams engage with key contribution steps could lead to better outcomes and a more effective open-source learning experience.

5.6 Benefits of collaboration in OSS projects

After contributing to OSS projects, we analyzed RQ_6 through student responses. Therefore, we performed a new qualitative analysis to identify the main benefits of collaborating on OSS projects. From the responses collected, we identified four categories of benefits most cited by students: (i) code improvement, (ii) helping maintain a more active community, (iii) improving students' knowledge, and (iv) providing faster and more innovative resolution of project problems. Table 22 presents the categories identified after analyzing the student responses. The first column lists the categories found, the second contains the number of students who experienced the benefits from contributing to OSS projects, and the third column contains the total number of students who experienced the respective benefit.

Table 22. Benefits of collaboration in OSS projects

Categories	Students	Total
Code improvement	P1, P5, P8, P10, P12, P13, P15, P20, P21, P22, P24, P27, P29	13
Help maintain a more active community	P2, P4, P9, P18, P21, P22, P28	7
Improve students knowledge	P3, P4, P9, P18, P19, P28	6
Provide faster and more innovative resolution of project issues	P1, P11, P12, P21	4

In Table 22, we can observe that thirteen students experienced an improvement in their code after contributing to the OSS project. This data suggests that the process of refactoring code smells in OSS projects was successful. Several reports support this assertion, as follows:

P1: "The most notable benefit for me was the assistance of various perspectives collaborating to identify and enhance codes."

P5: "The benefit of being an OSS project is that people can make these small refactoring, thus improving the code."

We also identified through the information in Table 22 that some students fall into more than one benefit category when contributing to OSS projects. As a result, we can see relationships between the categories of benefits that students encounter. The relationship we highlight is that the more contact students have with the community, the more they tend to see an increase in their knowledge. Some reports from students reinforce this statement:

P18: "I believe this helps an entire community and also helps you develop as a programmer."

P28:"Very cool, because you can actively participate in a project that has many people collaborating, engage the community more, and learn more."

Implications of RQ_6. Our findings indicate that the more a student sees the benefit of contributing to the community, the more they realize their own evolution as programmers. The correlation between communication with other developers and a feeling of improvement in students is noticeable. Therefore, it is important to highlight the importance of establishing communication with other programmers through OSS project communities. Furthermore, improving the code of OSS projects should also be highlighted as a big step for the students' professional careers. As a result, encouraging student contributions to OSS projects can be suitable for training professionals increasingly accustomed to communicating in the software development environment.

5.7 Students' perception of the contribution process in OSS projects

We examined RQ_7 by analyzing student feedback following their involvement in OSS projects. Following this, we conducted a new qualitative analysis to identify the primary challenges faced by students in contributing to such projects. From the responses, we identified four recurring categories of difficulties reported by students: (i) comprehending the project, (ii) initiating a pull request, (iii) grasping the contribution guidelines, and (iv) elucidating the issue addressed by the student. Table 23 showcases these categories based on our analysis of student responses. The first column lists the identified categories; the second column indicates the students who encountered difficulties contributing to OSS projects within each category; the third column denotes the total number of students facing the respective difficulty.

Table 23. Students' perception of the contribution process in OSS projects

Categories	Students	Total
Comprehending the project	P3, P6, P7, P8, P9, P14, P20, P21, P25, P26	10
Initiating a pull request	P1, P4, P5, P23, P27	5
Grasping the contribution guidelines	P15, P16, P18, P27	4
Elucidating the issue addressed by the student	P2, P10, P24	3

Table 23 shows that ten students had difficulty understanding the project, which became an obstacle to contributing to the OSS project. This fact suggests that OSS projects should provide a minimum level of project documentation to facilitate contributions from other developers. Some reports corroborate this statement, as follows:

P26:"There was a lack of explanatory documentation and internal support."

P21:"Code quality issues were often not simple and required in-depth analysis to identify the root cause."

Implications of RQ_7: Our findings underscore the relationship between the comprehensibility of a project and its collaborative potential. The more complex a project is to grasp, the more challenging it becomes to foster effective collaboration and enact meaningful changes. This correlation between project complexity and collaborative difficulty is unmistakable. Therefore, it becomes imperative to emphasize the necessity of enhancing project understanding through streamlined documentation. We enable smoother collaboration and more informed contributions by providing developers with more precise insights into project intricacies. Moreover, offering explicit guidance on how individuals can contribute to the project is vital. This ensures that newcomers can seamlessly integrate into the development process. Consequently, to augment the quantity and elevate the quality of contributions to OSS endeavors, it is essential to establish a minimum documentation outlining the project structure, alongside a comprehensive guide that delineates the contribution process.

6 Discussion

Our findings highlight several challenges students face when refactoring code smells in OSS projects. Firstly, it was identified that a lack of understanding of the source code directly impacts students' safety when carrying out refactoring. Many students reported difficulties reading and interpreting the code, resulting in hesitation to modify more complex components. This reinforces the need for educational strategies that emphasize identifying code smells and techniques for understanding the structure and architecture of software before refactoring.

Additionally, the analysis of the impact of refactoring on internal quality attributes yielded mixed results. While some refactoring reduced code complexity, others had a negative effect on attributes such as cohesion and coupling. These findings suggest that the teaching of refactoring should incorporate a more in-depth focus on quality metrics, helping students to evaluate the effects of their changes. Automated analysis tools can play an essential role in this process, providing immediate feedback on the changes made. The incremental refactoring approach applied in the study proved to be effective in mitigating the challenges faced by the students. By refactoring one code smell at a time and recording its impact on the quality attributes in the journal, the students could progressively evaluate the effects of their modifications. This practice enabled more refined control over the code's evolution, reducing the introduction of new code smells and making it easier to identify problems generated during the process. This methodology also fostered student autonomy, encouraging an iterative and reflective approach to continuous software improvement.

Another point identified was the students' insecurity when refactoring involving multiple files and interdependent components. The complexity of the modifications was a factor that generated fear, mainly when the refactoring affected several parts of the system. Based on this finding, it can be inferred that pedagogical practices that encourage incremental refactoring are essential, allowing students to make small, safe changes before tackling larger, more structural modifications.

Students' challenges when contributing to OSS projects point to the need for clear guidelines and accessible docu-

Table 24. Guidelines on teaching code smell refactoring and OSS projects

Guidelines on teaching code smells refactoring		
ID	**Description**	**Guideline**
G1	Teaching code reading techniques	Before starting refactoring, students should be trained to understand the structure of the source code, using diagrams and dependency visualization tools.
G2	Using metrics to evaluate refactoring	Incorporate the analysis of software quality metrics into teaching so that students can objectively evaluate the impact of their changes.
G3	Incremental refactoring	Encourage students to carry out small refactoring followed by automated tests, ensuring that the system's behavior remains unchanged.
G4	Simulation of real scenarios	Present challenges based on real code smells problems and lead students through the entire refactoring process, from identification to quality analysis.
Guidelines for using OSS projects in education		
G5	Strategic choice of projects	Select OSS projects with well-structured documentation and good contribution practices, facilitating student learning and engagement.
G6	Continuous evaluation of the student experience	Collect feedback from students on their difficulties and perceptions when working with OSS, adjusting the pedagogical approach as necessary.
G7	Facilitate contact with maintainers	Whenever possible, encourage students to interact with the project leaders, providing constructive feedback on their contributions.

mentation. Many students struggled to understand the contribution guidelines and review flow, which hindered the acceptance of their refactoring. This highlights the importance of detailed instructions on navigating an OSS project, from understanding the code base to submitting pull requests.

Given these challenges, it is essential to formulate guidelines that can guide both the teaching of code smell refactoring and the use of OSS projects as a pedagogical tool. The guidelines presented in Table 24 have been developed to provide more structured support for students, helping them to develop practical skills and overcome the difficulties they reported during the study.

Implementing these guidelines (see Table 24) can significantly improve the teaching of code smell refactoring, making students more confident and prepared to work on real projects, both academic and professional. In particular, applying G1 and G2 can improve the understanding and evaluation of code changes, while G3 and G4 ensure a gradual and controlled refactoring process. In the context of OSS projects, guidelines G5, G6, and G7 encourage more structured and collaborative participation, promoting a continuous learning environment in line with industry practices.

To complete this discussion, in the following sections, we present a **summary of the qualitative analyses** conducted based on student feedback and the evaluation of their contributions to OSS projects. These analyses enabled us to identify the primary challenges faced, perceptions about the contribution process, and the impact of refactoring activities on code comprehension and overall learning.

6.1 Challenges

One of the most common challenges reported by students was understanding the source code of the projects, as previously shown in Table 17 and Table 23. Many participants noted that unfamiliarity with the application domain and the absence of clear, up-to-date documentation made understanding the system's structure and logic a time-consuming and sometimes frustrating process. Projects with complex or non-standard architectures required a high cognitive effort, which directly impacted students' self-confidence and reduced their efficiency in performing refactoring tasks. In some cases, this difficulty led to hesitation in proposing changes or submitting pull requests, highlighting the importance of understanding the code as a determining factor for engagement and the success of contributions.

6.2 Perceptions of the Contribution Process

The analysis of qualitative feedback revealed that contributing to OSS projects also presented a significant obstacle for students (see Table 23). Many reported difficulties in understanding the workflow required by the projects, including reading the contribution guidelines, properly using version control tools (such as Git), and executing the pull request submission process. The lack of detailed documentation and inconsistency in contribution instructions were cited as factors that increased participants' uncertainty. This documentary barrier not only made it difficult to join projects, but also limited learning about collaborative practices and standards adopted in free software communities. Thus, clarity and accessibility of information proved to be crucial elements for the success of training activities.

6.3 Learning and Professional Development

Despite the difficulties reported, students recognized significant advances in terms of technical learning and professional development; these perceptions were previously presented in Table 11 and Table 10. Participation in OSS projects con-

tributed to the improvement of code reading and manipulation skills, as well as to a better understanding of the dynamics of collaborative work in real software development environments. The reports indicated that involvement in refactoring tasks provided a more practical view of system maintenance and evolution, strengthening skills related to code quality and technical communication. However, many students emphasized that a lack of deep understanding of the code base limited the effectiveness of refactorings, highlighting the importance of pedagogical strategies to support the learning of complex systems, such as guided reviews, dependency mapping, and mentoring by more experienced mentors.

6.4 Suggestions and Expectations

The suggestions collected reinforce the need for structural and pedagogical improvements in the process of integrating students into OSS projects. Participants expressed interest in having more detailed guidelines, practical examples, and ongoing support resources, such as tutorials, mentoring sessions, and peer discussion spaces. These demands reflect not only the search for greater autonomy, but also the perception that the quality of documentation and institutional support play a fundamental role in engagement and the consolidation of learning. Thus, the analyses indicate that strengthening project documentation and creating structured support mechanisms can amplify the positive impact of refactoring activities, promoting more meaningful and inclusive learning experiences.

7 Threats to Validity

We discuss potential threats to the validity of our study based on the classifications proposed by Wohlin et al. (2012).

Internal Validity. The students' limited experience with refactoring concepts, tools, and code smells is a potential threat to internal validity. We provided six weeks of training during the Software Quality course to address this issue. Another threat concerns the quality of the refactorings applied. We were unable to perform a comprehensive code review of all student submissions, which limits our ability to verify whether each code smell was addressed using the most suitable refactoring technique.

Construct Validity. A potential threat lies in interpreting students' responses to the post-activity questionnaire. Since the analysis includes qualitative feedback about their learning experience, some students may have hesitated to provide candid responses due to concerns about evaluation or judgment. We addressed this risk by explicitly encouraging students to respond sincerely and reassuring them that their answers would not affect their grades.

External Validity. Our findings are limited to Java-based object-oriented systems and React applications written in TypeScript. The results may not generalize to other programming paradigms, languages, or frameworks. Additionally, some variation may arise due to differences in project domains. Another limitation relates to the participants' varying levels of development experience and familiarity with code smells or refactoring. While we provided theoretical training and preparatory materials, each student's background may have influenced their performance. Finally, since the study was conducted in only two undergraduate courses at a single institution, with a specific group of students, the generalizability of our results to broader educational or professional contexts is limited.

8 Conclusion and Future Work

Our study investigated teaching Software Engineering students code smell refactoring practices through contributions to OSS projects and the impact of code refactoring on internal quality attributes. We consider Java and React projects, 20 types of code smell, and five internal quality attributes: cohesion, complexity, inheritance, coupling, and size. A total of 29 students, divided into 22 teams, refactored code smells in OSS projects as part of software quality and maintenance courses.

Our main results were: (i) students recognized improvements in code quality after refactoring; (ii) they identified strong connections between refactoring, testing, and debugging; (iii) their confidence decreased when refactoring required changes across multiple files; (iv) code complexity posed a significant challenge to refactoring; (v) students' choices of refactoring techniques were influenced by project structure and personal preferences, often combining multiple techniques to address a single smell; (vi) in some cases, refactoring introduced new code smells; (vii) the longest refactoring efforts were also the most likely to reintroduce code smells; (viii) contributing to OSS projects improved students' programming skills and fostered a sense of professional growth; (ix) students faced challenges in understanding OSS contribution processes, particularly regarding issue resolution, adherence to contribution guidelines, and responding to maintainer feedback; (x) automated checks and review workflows varied across projects, affecting students' ability to submit successful contributions; and (xi) despite these challenges, engagement with OSS enabled students to gain practical experience in collaborative software development. Our findings provide valuable perspectives for software engineering educators aiming to integrate refactoring practices into coursework while leveraging OSS contributions as an educational tool.

As future work in the context of teaching code smell refactoring practices, several research directions can be pursued to deepen our understanding and improve instructional approaches. First, we suggest employing additional tools to detect a broader range of code smells, including those not covered in this study, particularly those relevant to modern frameworks such as React. Second, automated refactoring tools could be investigated to assess their effectiveness in removing code smells and analyzing the impact on internal quality attributes. Third, future studies could isolate individual code smells to evaluate the specific effect of their refactoring on internal quality attributes, allowing for more granular analysis. Finally, experimental designs involving comparison groups, such as traditional classroom-based activities or gamified approaches, could help assess the added value of engaging students in real-world open-source contributions despite the inherent challenges of implementing such studies.

Replication Package and Data Availability

All the materials used in this study are available in a replication package to ensure transparency and enable replication. The package includes the questionnaires used before and after the intervention, the methodology description, the reflection diary template, spreadsheets with students' answers and analyses, and summaries of the soft and hard skills gained. It also includes documents related to software quality analysis in both `Java` and `React` projects. The complete package can be accessed at: https://doi.org/10.5281/zenodo.11087486

Acknowledgements

This work is partially supported by the Cearense Foundation of Scientific and Technological Support (FUNCAP) grant BP5-00197-00042.01.00/22, the National Council for Scientific and Technological Development (CNPq) grant 404406/2023-8, and the São Paulo Research Foundation (FAPESP) and the São Paulo State Data Analysis System Foundation (SEADE), Process Number 2023/18026-8.

References

Aberdour, M. (2007). Achieving quality in open source software. *IEEE Software*, 24(01):58–64.

Agrahari, V. and Chimalakonda, S. (2020). Refactor4green: a game for novice programmers to learn code smells. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 324–325. IEEE.

Al Dallal, J. and Abdin, A. (2018). Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69.

AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2024). Automating source code refactoring in the classroom. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 60–66, New York, NY, USA. Association for Computing Machinery.

Berg, A., Osnes, S., and Glassey, R. (2022). If in doubt, try three: Developing better version control commit behaviour with first year students. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 362–368, New York, NY, USA. Association for Computing Machinery.

Bezerra, C., Alves, V., Lobo, A., Queiroz, J., Lima, L., and Meirelles, P. (2024). Contributing to open-source projects in refactoring code smells: A practical experience in teaching software maintenance. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*, pages 399–409, Porto Alegre, RS, Brasil. SBC.

Bezerra, C., Damasceno, H., and Teixeira, J. (2022). Perceptions and difficulties of software engineering students in code smells refactoring. In *Anais do X Workshop de Visualização, Evolução e Manutenção de Software*, pages 41–45, Porto Alegre, RS, Brasil. SBC.

Braun, V. and Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Destefanis, G., Counsell, S., Concas, G., and Tonelli, R. (2014). Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*, pages 157–170. Springer.

Diniz, G. C., Silva, M. A. G., Gerosa, M. A., and Steinmacher, I. (2017). Using gamification to orient and motivate students to contribute to oss projects. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 36–42.

Dong, J. Q. and Götz, S. J. (2021). Project leaders as boundary spanners in open source software development: A resource dependence perspective. *Information Systems Journal*, 31(5):672–694.

dos Santos, H. M., Durelli, V. H. S., Souza, M., Figueiredo, E., da Silva, L. T., and Durelli, R. S. (2019). Cleangame: Gamifying the identification of code smells. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, page 437–446, New York, NY, USA. Association for Computing Machinery.

Ellis, H. J. C., Hislop, G. W., Chua, M., Kussmaul, C., and Burke, M. M. (2010). Panel — teaching students to participate in open source software projects. In *2010 IEEE Frontiers in Education Conference (FIE)*, pages F2B–1–F2B–2.

Farchi, E., Nir, Y., and Ur, S. (2003). Concurrent bug patterns and how to test them. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 7 pp.–.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, New York, NY, USA. Association for Computing Machinery.

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in react-based web apps. *Information and Software Technology*, 155:107111.

Fowler, M. (2018). *Refactoring: improving the Design of Existing Code*. Addison-Wesley Professional.

França, C., da Silva, F. Q. B., and Sharp, H. (2020). Motivation and satisfaction of software engineers. *IEEE Transactions on Software Engineering*, 46(2):118–140.

Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T., and Mkaouer, M. W. (2021). One thousand and one stories: A large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1303–1313, New York, NY, USA. Association for Computing Machinery.

Haider, S., Khalil, W., Al-Shamayleh, A. S., Akhunzada, A.,

and Gani, A. (2023). Risk factors and practices for the development of open source software from developers' perspective. *IEEE Access*, 11:63333–63350.

Halepmollasi, R. and Tosun, A. (2024). Exploring the relationship between refactoring and code debt indicators. *Journal of Software: Evolution and Process*, 36(1):e2447.

Hoffmann, M., Nagle, F., and Zhou, Y. (2024). The value of open source software. *Harvard Business School Strategy Unit Working Paper*, (24-038).

Hu, Z., Song, Y., and Gehringer, E. F. (2019). A test-driven approach to improving student contributions to open-source projects. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–9.

Inc., S. T. (2023). Understand software metrics.

Jiang, J.-Y., Cheng, P.-J., and Wang, W. (2017). Open source repository recommendation in social coding. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, page 1173–1176, New York, NY, USA. Association for Computing Machinery.

Jin, Y., Bai, Y., Zhu, Y., Sun, Y., and Wang, W. (2023). Code recommendation for open source software developers. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 1324–1333, New York, NY, USA. Association for Computing Machinery.

Kaur, A., Jain, S., Goel, S., and Dhiman, G. (2021). A review on machine-learning based code smell detection techniques in object-oriented software system(s). *Recent Advances in Electrical Electronic Engineering*, 14(3):290–303.

Keuning, H., Heeren, B., and Jeuring, J. (2019). How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 119–125, New York, NY, USA. Association for Computing Machinery.

Keuning, H., Heeren, B., and Jeuring, J. (2021). A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 562–568, New York, NY, USA. Association for Computing Machinery.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

Mariani, T. and Vergilio, S. R. (2017). A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34.

Martins, J., Bezerra, C., Uchôa, A., and Garcia, A. (2020). Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 52–61, New York, NY, USA. Association for Computing Machinery.

Martins, J., Bezerra, C., Uchôa, A., and Garcia, A. (2021). *How Do Code Smell Co-Occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective*, page 54–63. Association for Computing Machinery, New York, NY, USA.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320.

Menolli, A., Strik, B., and Rodrigues, L. (2024). Teaching refactoring to improve code quality with chatgpt: An experience report in undergraduate lessons. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*, SBQS '24, page 563–574, New York, NY, USA. Association for Computing Machinery.

Morgan, B. and Jensen, C. (2014). Lessons learned from teaching open source software development. In Corral, L., Sillitti, A., Succi, G., Vlasenko, J., and Wasserman, A. I., editors, *Open Source Software: Mobile Open Source Technologies*, pages 133–142, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nanthaamornphong, A. and Boonchieng, E. (2023). An exploratory study on code smells during code review in oss projects: A case study on openstack and wikimedia. *Recent Advances in Computer Science and Communications*, 16(7):20–33.

Nanthaamornphong, A., Saeang, T., and Tularak, P. (2020). Zsmell – code smell detection for open source software. *International Journal on Advanced Science, Engineering and Information Technology*, 10(3):1035–1041.

Nascimento, D. M., Cox, K., Almeida, T., Sampaio, W., Bittencourt, R. A., Souza, R., and Chavez, C. (2013). Using open source projects in software engineering education: A systematic mapping study. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 1837–1843.

Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., and Zhao, Y. (2016). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 440–451, New York, NY, USA. Association for Computing Machinery.

Pinto, G., Ferreira, C., Souza, C., Steinmacher, I., and Meirelles, P. (2019). Training software engineers using open-source software: The students' perspective. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 147–157.

Qiu, H. S., Lieb, A., Chou, J., Carneal, M., Mok, J., Amspoker, E., Vasilescu, B., and Dabbish, L. (2023). Climate coach: A dashboard for open-source maintainers to overview community dynamics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA. Association for Computing Machinery.

Silva, J. O., Wiese, I., German, D. M., Treude, C., Gerosa, M. A., and Steinmacher, I. (2020). Google summer of code: Student motivations and contributions. *Journal of Systems and Software*, 162:110487.

Sousa, L., Oliveira, A., Oizumi, W., Barbosa, S., Garcia, A., Lee, J., Kalinowski, M., de Mello, R., Fonseca, B.,

Oliveira, R., Lucena, C., and Paes, R. (2018). Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 921–931, New York, NY, USA. Association for Computing Machinery.

Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., and Counsell, S. (2018). Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, EASE'18, page 68–78, New York, NY, USA. Association for Computing Machinery.

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

Yamashita, A. and Moonen, L. (2013). Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251.

Yu, Y., Wang, H., Yin, G., and Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218.