Contributing to open-source projects in refactoring code smells: A practical experience in teaching Software Maintenance

Carla Bezerra
carlailane@ufc.br
Federal University of Ceará
Quixadá, Brazil

Victor Anthony Alves
victorpa@alu.ufc.br
Federal University of Ceará
Quixadá, Brazil

Antônio Hugo Lobo
hugorplobo@alu.ufc.br
Federal University of Ceará
Quixadá, Brazil

João Paulo Queiroz
Joaop3595@gmail.com
Federal University of Ceará
Quixadá, Brazil

Lara Lima
laragabriellysouzabatista@gmail.com
Federal University of Ceará
Quixadá, Brazil

Paulo Meirelles
paulormm@ime.usp.br
University of São Paulo
São Paulo, Brazil

ABSTRACT

Code smells are inadequate code structures that can harm quality and maintainability. To remove these deficient structures, developers use refactoring techniques. Refactoring helps code be easier to understand and modify by eliminating potential problems and improving internal quality attributes. Most refactoring activities are usually performed manually and undisciplined, which can cause code degradation. Concepts, practices, software refactoring tools, and code smells are rarely discussed in undergraduate computing courses. This problem is reflected in the software industry, which generally does not use refactoring practices to improve code readability and maintainability. In this context, we present in this paper an experience report on teaching the practice of code smell refactoring and the impact on internal quality attributes through contribution to Open Source Software (OSS) projects. The study was carried out in two undergraduate classes in Software Quality and Software Maintenance courses, and our main results were that: (i) students observed improvements in code quality after refactoring smells; (ii) they noted connections between refactoring, testing, and debugging; (iii) they felt less confident refactoring code spread across multiple files; (iv) code complexity hindered their ability to refactor; (v) the choice of refactoring techniques depended on factors like project structure and personal preference, with techniques often used in combination to address a single smell; (vi) most refactorings decrease internal quality attributes; (vii) contributing to OSS projects fostered a sense of programmer growth; and, (viii) project clarity was linked to its potential for collaboration.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Open source model**; • **General and reference → Empirical studies**; • **Applied computing → Education**.

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 [18]. These anomalies can affect any software system [38]. Unlike a bug or defect in the software code, the presence of code smells does not mean the presence of defects in the software. However, these anomalies can have other negative consequences, impacting the software maintenance and evolution [29]. Refactoring can remove code smells and can have a direct impact on code quality [47]. Fowler [18] defines refactoring as a set of small changes that are made to the internal structure of the code without changing the external behavior. In practice, the objectives of refactoring may vary. Some of these objectives include combating software design degradation, reducing the effort to perform maintenance activities, facilitating the implementation of new features, fixing bugs present in the system, and removing code smells [19, 21, 31].

Social coding and Open Source Software (OSS) have experienced a surge in popularity, offering software developers diverse avenues for community engagement and collaborative work [24]. This paradigm allows external contributors to suggest modifications to a software project without necessitating access to the central repository held by the core team [48]. High-quality OSS projects rely heavily on a large and sustainable community to develop quality code, debug the code effectively, and create new features [1]. Recent studies show that development in OSS projects can entail risks related to the quality and maintenance of the software [6, 20, 25]. One of the main risk factors influencing the health of OSS projects is directly linked to the maturity of the development processes, especially concerning the quality of the code submitted by contributors [40]. Among these quality deficits, one most commonly identified is code smells [36]. For OSS maintainers, it is both challenging and time-consuming to find competent contributors within a potentially large candidate pool [25].

In the educational context, software engineering instructors use OSS projects as part of their assessment model, exposing many perceived benefits in student knowledge [39]. The participation of students in the collaboration of open source projects has been the subject of several studies [11, 37], which pointed out that students' skills and knowledge can make a significant and positive contribution to these projects. Among the most promising skills, refactoring code smells and resolving issues are the most highlighted by the students [7]. However, although students demonstrate these skills, there is often a lack of motivation for them to take an active part and get involved in improving the quality of real projects [42]. It is, therefore, essential to involve students in contributing to OSS projects, allowing them to apply their skills to improve the quality

of these projects [39]. This approach not only directly benefits OSS projects but also provides students with a valuable opportunity for hands-on learning and the development of essential skills for their careers in the field of information technology [11, 13, 22].

This paper presents an experience report of the teaching and practice of code smell refactoring in Java and React OSS projects. The experiment was carried out with two Software Quality and Maintenance classes, analyzing the perceptions of 29 students. The students identified and refactored 10 React code smells and 10 Java code smells in 23 OSS projects. All 23 projects received refactoring contributions and improved code quality, and most students made their first contributions to OSS projects. We analyzed the following aspects in our study: (i) students' perception of refactoring practices; (ii) code smells that are harder to refactor and difficulties in refactoring code smells; (iii) refactoring techniques most used to remove code smells; (iv) impact of code smells refactoring on internal quality attributes; (v) benefits of collaboration in OSS projects; and, (vi) students' perception of the contribution process in OSS projects. Our report and methodology can help software engineering instructors carry out practices with students to improve code quality and contribute to OSS projects.

2 BACKGROUND

Code smells may indicate design problems at multiple granularity levels, i.e., design problems at method and class levels. Software developers often rely on code smells as indicators of code quality [18, 43]. For instance, developers have often used tools like Stack Overflow to ask about code smells and anti-patterns and identify these anomalies in their source code [45]. There are several tools for detecting code smells [16]. In this study, two tools were used to analyze code smells: `PMD`[1], for detecting code smells in OSS Java projects, and the `React Sniffer`[2] extension of a tool also called `React Sniffer`[3], aimed at detecting code smells in `React` projects that specifically use the `Typescript` language. Several studies have used such tools in the literature [15, 17, 32, 33]. The Table 1 and the Table 2 show, respectively, the code smells found by `PMD` and `React Sniffer` that will be considered in this study.

Table 1: Code smells detected by PMD

Code Smells	Description
Excessive Method Length	A method with a high number of lines of codes [28]
Excessive Class Length	A class with a high number of lines of codes [28]
Double Checked Locking	An object is initialized but not all object fields are necessarily written to the heap [14]
Duplicated Code	Identical or very similar pieces of code [18]
God Class	Too many software features in a class. It tends to be very large and hard to read and understand [18]
Data Class	Classes that have fields, getting and setting methods for the fields, and nothing else [18]
Collapsible If Statements	A series of nested 'if' statements [5]
Excessive Parameter List	Function or method that have too many parameters [18]
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

The OSS development model has attracted the attention of various communities, including companies, professionals, and researchers [1, 8]. This model allows developers to engage with communities, collaborate, and expand their knowledge [24]. The democratized nature of contributions to OSS projects attracts many developers who are not members of the core project team, who collaborate to add value to various areas of the project architecture [25, 35].

Table 2: Code smells detected by ReactSniffer

Code Smells	Description
Any Type	The use of the ‘any‘ type in TypeScript disables the of 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 [17]
Large File	Files with a lot of lines and components [17]
Large Component	Components that are difficult to read because they have a large number of lines of code [17]
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 [17]
Force Update	When a piece of code forces a component or page to reload[17]
Uncontrolled Component	A component that does not use props/state to handle form's data[17]

In the context of Software Engineering Education (SEE), access to the community, to the software source code and to information about its development and evolution play key roles that drive the use of OSS projects in academic activities [41]. These factors not only enrich student learning, but also provide valuable opportunities to explore the practical principles and challenges of software engineering in a real-world environment [44]. The contributions made by students have played a significant and beneficial role in OSS projects [37]. Among these contributions, adding new features and fixing problems are the most common, highlighting students' positive impact on the evolution and quality of these projects [39].

3 RELATED WORK

The studies by Keuning et al. [26, 27] address the teaching of code quality in software engineering education. In [26], the authors investigated how teachers approach code quality, identifying problems and providing guidelines for improving students' code quality. Based on teachers' guidance and code quality tools, they developed guidelines for refactoring, which, in this paper, we use as a basis for students to evaluate the improvement of refactored code. Then, Keuning et al. [27], they presented a tutoring system to allow students to practice improving code in small programs that are already functional. Based on rules obtained from studies with teachers and professional tools, this system defines how the code should be rewritten without altering its functionality. Our work used this system to teach students about refactoring practices.

Agrahari and Chimalakonda [2] introduced Refactor4Green, a game developed to promote the teaching of code smells and refactoring. Such a game focuses on code smells related to energy efficiency and, in short, contains learning cards with the definition of some code smells and their refactoring techniques, followed by challenges in the form of objective quizzes. Similarly, dos Santos [12] introduced the CleanGame, a gamified platform for practicing code smell identification. This platform has two major modules, one for students to review the main concepts about various code smells and another for practical tasks of identifying code smells

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

in the source code of Java systems. In both works, the evaluation carried out with students shows that gamification can be helpful for teaching code smells and refactoring. Our work relates to these because we also report on a practical teaching activity and want to understand if and how it contributes to the student's experience.

Pinto et al. [39] explore software engineering students' perspectives on using OSS projects in undergraduate courses. The authors conducted 21 semi-structured interviews with students to understand their experiences, challenges, and perceptions when contributing to OSS projects during the course. In addition, they discuss the importance of teacher involvement in influencing student participation in OSS projects. The study reveals students' positive perceptions of using OSS projects, with the majority of them positively evaluating this practice as enriching their experience and learning during their degree. The present work is directly related to this because we used these perceptions as motivators to carry out the contribution practice in OSS projects with the students.

AlOmar et al. [4] investigate aspects of the automation of source code refactoring in the classroom; the objectives include demonstrating how students apply the refactoring technique using a specific refactoring tool and investigating students' perceptions of the tool's usefulness, usability, and functionality. The methodology involved the analysis of refactoring submissions from students on software engineering courses, with an empirical approach to assess the quality of the refactored code and a qualitative approach through a student survey. The results indicated that tool-assisted refactoring positively impacted the correction of code antipatterns. In this study, we used a similar refactoring analysis method to collect the techniques and impacts of the refactoring carried out by the students on the OSS projects.

4 STUDY SETTINGS

This study aims to report the experience of the teaching process and practice of refactoring code smells in `Java` and `React` with contributions to OSS projects. It aims to analyze how eliminating these code smells affects internal quality attributes and contributes to OSS projects by removing these code smells.

4.1 Research Questions

We have defined the following research questions (**RQs**) to guide our research:

***RQ*$_1$** – *What is students' perception about the practice of code smells refactoring?* **RQ$_1$** seeks to understand the students' perception of the importance of the practice of code smell refactoring. By answering **RQ$_1$**, we can understand the students' opinion of employing code smell refactoring practices. Seeks to identify which code smell refactoring technique was used the most. The literature suggests refactoring techniques for some code smells [18]. However, for most code smells, the refactoring techniques used to remove them are free. We want to know if students are refactoring code smells appropriately.

***RQ*$_2$** – *What are the code smells that are harder to refactor according to students' perceptions?* **RQ$_2$** seeks to identify the types of code smells that are harder to refactor from the perspective of each student. To answer **RQ$_2$**, we quantitatively analyzed which code smells were identified by the students as the most difficult to refactor. Furthermore, we identified that the code smell refactoring technique was used the most. The literature suggests refactoring techniques for some code smells [18]. However, for most code smells, the refactoring techniques used to remove them are free. We want to know if students are refactoring code smells appropriately.

***RQ*$_3$** – *What are the difficulties of refactoring code smells according to students' perceptions?* By answering **RQ$_3$**, we can understand students' most common difficulties during refactoring practices. In addition, we can analyze the impact of refactoring code smells on internal quality attributes and understand if there is a relationship between the difficulties encountered and the outcome of the refactorings.

***RQ*$_4$** – *What are students' perceptions of the benefits of collaboration in OSS projects from the perspective of improving code quality?* After students submit refactorings as contributions to OSS projects, **RQ$_4$** aims to identify the benefits of improving code quality from the student's perspective. This improvement is qualitatively identified in the questionnaires answered by students and in submitting contributions to projects.

RQ*$_5$: What are students' perceptions of the contribution of refactorings in OSS projects?* At **RQ$_5$**, we aim to investigate students' perceptions of the contribution process in OSS projects. Contributions are made according to project standards by student teams and may be accepted or rejected by the project maintainers.

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 code smell refactoring. This phase encompassed three weeks of theoretical classes, totaling 12 hours, focusing 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, as described in Table 6), with practical sessions utilizing the `Understand` tool. Additionally, students participated in a workshop session discussing contributions to OSS software, spanning 2 hours. Concluding the curriculum component, one week, involving 4 hours, was allocated to instructing students on effectively utilizing `PMD` and `ReactSniffer` tools. This comprehensive educational program spans 26 hours, pivotal in enhancing 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

if 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.

Table 3: Students profile

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

Step 2: Presentation of the code smells refactoring practice. After passing the concepts and tools to the students, we presented the code smell refactoring practice tasks the participants should carry out, which consisted of the final work of the course. In teams of two people or individually, students would have to select an OSS project in Java or React that was available to choose from a list to identify and refactor the code smells of the project. Table 4 shows the division of students by class. Each team had to refactor at least four different types of code smells, wherein at least 20 occurrences of code smell of these four types would be refactored.

Table 4: Students divided by class

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

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 7 provides an overview of the selected projects, including project ID, team composition, students involved in refactoring, programming language, and total lines of code.

Table 5: 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

T2 - Identify and address code smells. Students were allowed to choose four types of code freely smells if the analysis tool detected more than four types. Using `PMD` or `ReactSniffer`, they identified and addressed 20 instances of these code smells. This approach empowered students to focus on the most prevalent or relevant code quality areas, enhancing their understanding of refactoring techniques and software design principles. Table 5 illustrates the number of refactored code smells documented in students' diaries and the 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 6: 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), e Bases Classes (IFANIN). By analyzing these metrics, students could observe improvements in various aspects of code quality, providing valuable insights into the effectiveness of the refactoring process.

T4 - Refactor one type of code smell at a time, documenting challenges and techniques. Following the initial analysis using Understand, students proceeded to refactor the identified code smells systematically, addressing one type at a time. Upon completing the refactoring process for each type, students were required to maintain a diary documenting the difficulties they encountered, the refactoring methods employed, and any observed changes, including the potential introduction of new code smells or improvements in quality metrics.

T5 - Conduct a code review and submit contributions, specifying improvements made based on metrics. During the last phase, students conducted code reviews of their refactored code and submitted contributions to the project repository. Based on gathered metrics, they specified improvements, addressed code quality enhancements, and resolved any remaining code smells. After completing the code

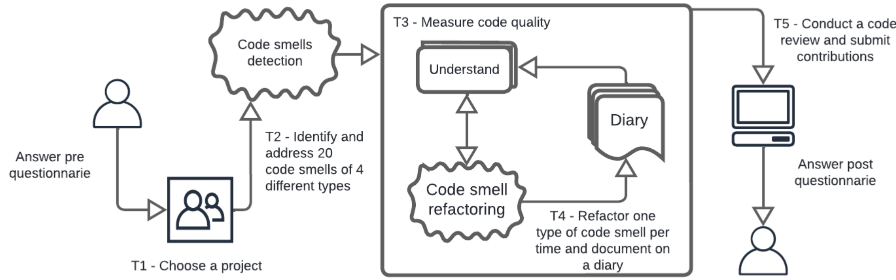

Figure 1: Tasks from practice delivery

Table 6: 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, more complex is the code structure [34].
	Sum Cyclomatic Complexity (SCC)	Sum of cyclomatic complexity of all nested functions or methods. The higher the value of this metric, more complex is the code structure [34].
	Average Cyclomatic Complexity (ACC)	Average cyclomatic complexity for all nested functions or methods. The higher the value of this metric, more complex is the code structure [34].
	Nesting (MaxNest)	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function. The higher the value of this metric, more complex is the code structure [30].
Size	CountDeclFunction	Number of functions. The higher the value of this metric, the larger the system size [23].
	CountLine	Number of physical lines. The higher the value of this metric, the larger the system size [23].
	Comment Lines of Code (CLOC)	Number of lines containing comment. The higher the value of this metric, the larger the system size [30].
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, less cohesive is the class [9].
Coupling	Coupling Between Objects (CBO)	Number of classes that a class is coupled. The higher the value of this metric, more coupling is the classes and methods [9].
Inheritance	Depth of Inheritance Tree (DIT)	Maximum depth of class in inheritance tree. The higher the value of this metric greater is the degree of inheritance of a system [9].
	Number Of Children (NOC)	Number of immediate subclasses. The higher the value of this metric, the greater the degree of inheritance of a system [9].
	Bases Classes (IFANIN)	Number of immediate base classes. The higher the value of this metric, the greater the degree of inheritance of a system [10].

smells refactoring task, students were asked to answer a questionnaire to analyze 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 the 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.

Finally, Figure 1 explains the workflow of the tasks. Additionally, we conducted a pre-questionnaire before the tasks' commencement and a post-questionnaire after completing the tasks.

5 RESULTS AND DISCUSSION

In the following sections, we show our findings and discuss our results, explicitly answering the research questions of this study.

Table 7: Project characterization

System	Teams	Students	Language	LOC
S1	T1	P27	Java	43145
S2	T2	P6, P21	Java	73542
S3	T3	P11	Java	5571
S4	T4	P29	Java	6129
S5	T5	P25, P7	Java	14313
S6	T6	P26, P2	Java	5126
S7	T7	P17	Java	7560
S8	T8	P13, P8	Java	16989
S9	T9	P14	Java	96523
S10	T10	P28	Java	114600
S11	T11	P1	React	17490
S12	T12	P4	React	58937
S13	T13	P9, P3	React	24317
S14	T14	P12	React	32058
S15, S16, S17	T15	P22	React	3870, 4893, 134996
S18	T16	P15, P24	React	28549
S19	T17	P10, P16	React	334060
S20	T18	P18	React	29723
S21	T19	P19	React	51042
S22	T20	P20	React	104717
S23	T21	P5	React	264719

5.1 Students' perception of refactoring practices

We addressed **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 8) and what soft and hard skills they acquired through the execution of the work (See Tables 9 and 10).

Table 8: 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

Observing Table 8, it is possible to notice that a considerable number of the 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 tend to positively impact the quality of

a system, according to the students' perceptions. Some student reports corroborate this statement:

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 9: Students' perceptions about the soft skills acquired with the code smells refactoring

Soft Skills	Students	Total
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

Analyzing Table 9, we noticed that several students (13) did not perceive any acquisition of soft skills after practicing code smell refactoring. It is important to emphasize that this does not mean that practicing refactoring does not lead to soft skill development, as these students may consider themselves experienced in the soft skills used during the work, such as creativity, problem-solving, and teamwork, which other students mentioned.

Table 10: 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

Finally, Table 10 shows us that the primary hard skills developed according to students' perceptions were debugging, testing, and refactoring techniques. This suggests an association between the use of tests and debugging tools as aids in the refactoring process, as proposed in the literature [18], 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."

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 address $\mathbf{RQ}_2$ by analyzing student responses after refactoring code smells. Thus, we conducted a new qualitative analysis to identify which code smells were the most difficult to remove. Out of the eight refactored code smells in the React projects, we highlight the code smells *Large Component* and *Too Many Props*. In Java projects, out of the fifteen refactored code smells, the code smells *God Class, Data Class*, and *Large Class* were found to be the most cited by students as the most difficult to refactor. Table 11 presents the most challenging code smells to refactor and the refactoring techniques students use.

Table 11: Code smells harder to refactor and the refactoring techniques used

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

From Table 11, we could see 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 [18] points out that refactoring code smells require 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, highlighting 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 required a cautious and well-planned approach to ensure the effectiveness of the improvements made by the students.

On the other hand, 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 deal with those more difficult code smells. These techniques aim to break down the code into smaller, more manageable parts, each with a clear responsibility [18]. 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 and because they are large and often complex files to refactor. In this way, it can be inferred that good programming practices and teaching focused on refactoring techniques facilitate the students' comprehension process. Based on these results, it is possible to draw up a document that will serve as a support to help students understand what each code smell is related to and the possible activities to remove them. In addition, it is interesting to see 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 Difficulties of refactoring code smells

To address $\mathbf{RQ}_3$, we conducted a qualitative analysis of student responses gathered after refactoring code smells. This analysis aimed to identify students' main difficulties during the refactoring process.

From the answers collected, we identified four categories of difficulties that most occurred among the students: (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 12 presents the categories identified after analyzing the students' responses. The first column lists the categories found; the second contains the number of students who had the respective difficulties caused during refactoring the code smell.

Table 12: Difficulties identified by students in the practice of refactoring

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

Examining Table 12, we noticed that eight students faced difficulties when refactoring a code smell without this, resulting in the emergence of bugs in the project. This 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 12 that some students are in more than one category of difficulties 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, in order 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: *cohesion, coupling, complexity*, and *inheritance* for Java projects (see Table 14), and *complexity* and *size* for React projects (see Table 13), their respective metrics, and the percentage change in each metric's value before and after refactoring for each system. To analyze the impact of code smell refactoring on attributes with more than one metric, we compared the sum of the pre- and post-refactoring metrics using the data collected from the Understand tool. The symbol ↑ represents an increase in the metric's value, the symbol ↓ represents a decrease in the metric's value, and the symbol – shows that the attribute value has not changed after refactoring the code smells. It is important to note that if the cohesion value increases, this attribute has been improved due to the greater cohesion of a class or method, thus improving 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 for three projects, improved for only one project, and remained unchanged for another. Coupling had mixed results, while project size and inheritance remained the same. However, complexity was reduced in four projects and remained the same in one project, with significant reductions in two projects (S8 with 25.13% and S3 with 10.90%). These two projects with the most significant 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 13, 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 [3], 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 refactorings may result from students' difficulties during the refactoring process.

Table 13: 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 14: 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%**

Implications of RQ_3.: Our findings indicate that even after training, students still encountered various difficulties while refactoring code smells. 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 refactorings revealed mixed results. There were mixed outcomes in Java projects, including cohesion degradation in some cases. In React projects, there were slight overall increases in complexity and size. This corroborates the idea that refactoring code smells may yield no results or even lead to negative outcomes, which, in this case, we can theorize is a product of the difficulties students encounter.

5.4 Benefits of collaboration in OSS projects

After contributing to OSS projects, we analyzed $\mathbf{RQ}_4$ 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 15 presents the categories identified after analyzing the student responses. The first column lists the categories found, the second contains the number of students who felt the benefits caused when contributing to OSS projects, and the third column contains the total number of students who felt the respective benefit.

Table 15: 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 15, we can observe that thirteen students handled 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 refactorings, thus improving the code."

We also identified through the information in Table 15 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 with the community, the more students 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 $\mathbf{RQ}_4$. Our findings indicate that the more the student sees the benefit of contributing to the community, the more he realizes his evolution as a programmer. 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 good for training professionals who are increasingly accustomed to communicating in the software development environment.

5.5 Students' perception of the contribution process in OSS projects

We investigated $\mathbf{RQ}_5$ by analyzing student feedback following their involvement in OSS projects. Afterward, we conducted a new qualitative analysis to pinpoint the primary challenges encountered 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 16 showcases these categories based on our analysis of student responses. The first column enumerates the identified categories; the second column indicates the number of students encountering difficulties contributing to OSS projects within each category; the third column denotes the total count of students facing the respective difficulty.

Table 16: 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

Examining Table 16, we noticed that ten students had difficulty understanding the project, which became an obstacle to contributing to the OSS project. This fact suggests the need for OSS projects to provide a minimum 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 $\mathbf{RQ}_5$: Our findings underscore the relationship between the comprehensibility of a project and its collaborative potential. Essentially, 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 indispensable to establish minimum documentation outlining the project structure alongside a comprehensive guide delineating the contribution process.

6 THREATS TO VALIDITY

We discuss threats to the study validity [46] as follows.

Internal Validity. One of the threats to internal validity is the student's lack of experience with refactoring concepts and tools and code smells. We provided them with a six-week training in the Software Quality course to mitigate this threat. Another threat is related to the quality of refactorings. It was impossible to code review all projects to ensure that the refactorings applied to the code smells were the most appropriate.

Construct Validity. A construct threat validity of the questionnaire due to some research questions in the study, the construct threat validity of the questionnaire involved a qualitative analysis of students perceptions of the practice of refactoring code smells which may lead students to hesitate in giving a truthful answer due to the apprehension of being evaluated. We tried to emphasize that the ideal was to be as honest as possible with the answer to mitigate this problem.

External Validity. The results apply solely to Java-based object-oriented systems and those developed using the React library. A limitation pertains to the system domains, where varying results may arise from different domains. Another problem we identified is that some students have little development experience or knowledge of code smells, software refactoring, or quality metrics. We provided all students with training and three weeks of theoretical classes on code smells to overcome this obstacle.

7 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 software maintenance courses.

Our main results were: (i) in the students' perception, the refactoring of code smells improved the quality of the system; (ii) there is a relationship between testing and debugging activities with the refactoring process; (iii) students do not feel safe refactoring code smells that involve multiple files; (iv) the harder it is to understand the code, the harder it is to refactor code smells; (v) the choice of a refactoring technique can be guided by several factors, from the project architecture to personal interests, and they can be used in a complementary way to remove a code smell; (vi) most of the refactorings worsened the internal quality attributes; (vii) the contribution to OSS projects is related to the students' vision of evolution as a programmer; and, (viii) the comprehensibility of a project correlates with its collaborative capacity.

As future work in the teaching of code smells refactoring practice, we should consider (i) employing other tools to detect other code smells that were not included in the study (especially for React), (ii) using automatic refactoring tools to remove code smells and analyze the impact on internal quality attributes; and (iii) comparing the impact of refactoring each code smell on internal quality attributes independently.

ARTIFACT AVAILABILITY

We provide our data and artifacts under open licenses at: https://zenodo.org/records/13010596

REFERENCES

[1] M. Aberdour. 2007. Achieving Quality in Open Source Software. *IEEE Software* 24, 01 (jan 2007), 58–64. https://doi.org/10.1109/MS.2007.2
[2] Vartika Agrahari and Sridhar Chimalakonda. 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)*. IEEE, 324–325. https://doi.org/10.1145/3377812.3390792
[3] Jehad Al Dallal and Anas Abdin. 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 (2018), 44–69. http://10.1109/TSE.2017.2658573
[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2024. Automating Source Code Refactoring in the Classroom. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (<conf-loc>, <city>Portland</city>, <state>OR</state>, <country>USA</country>, </conf-loc>) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3626252.3630787
[5] J Anderson. 2020. Addressing novice coding patterns: Evaluating and improving a tool for code analysis and feedback. *Report UUCS-20-002, University of Utah, Tech. Rep.* (2020).
[6] Amanda Berg, Simon Osnes, and Richard Glassey. 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* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 362–368. https://doi.org/10.1145/3478431.3499371
[7] Carla Bezerra, Humberto Damasceno, and João Teixeira. 2022. Perceptions and Difficulties of Software Engineering Students in Code Smells Refactoring. In *Anais do X Workshop de Visualização, Evolução e Manutenção de Software* (Online). SBC, Porto Alegre, RS, Brasil, 41–45. https://doi.org/10.5753/vem.2022.226804
[8] A. Capiluppi, P. Lago, and M. Morisio. 2003. Characteristics of open source projects. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 317–327. https://doi.org/10.1109/CSMR.2003.1192440
[9] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895
[10] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. 2014. Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*. Springer, 157–170. https://doi.org/10.1007/978-3-319-06862-6_11
[11] Guilherme C. Diniz, Marco A. Graciotto Silva, Marco A. Gerosa, and Igor Steinmacher. 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)*. 36–42. https://doi.org/10.1109/CHASE.2017.7
[12] Hoyama Maria dos Santos, Vinicius H. S. Durelli, Maurício Souza, Eduardo Figueiredo, Lucas Timoteo da Silva, and Rafael S. Durelli. 2019. CleanGame: Gamifying the Identification of Code Smells. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES 2019)*. Association for Computing Machinery, New York, NY, USA, 437–446. https://doi.org/10.1145/3350768.3352490
[13] Heidi J. C. Ellis, Gregory W. Hislop, Mel Chua, Clif Kussmaul, and Matthew M. Burke. 2010. Panel — Teaching students to participate in Open Source Software projects. In *2010 IEEE Frontiers in Education Conference (FIE)*. F2B–1–F2B–2. https://doi.org/10.1109/FIE.2010.5673437
[14] E. Farchi, Y. Nir, and S. Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings International Parallel and Distributed Processing Symposium*. 7 pp.–. https://doi.org/10.1109/IPDPS.2003.1213511
[15] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and*

Software Technology 126 (2020), 106347. https://doi.org/10.1016/j.infsof.2020.106347

[16] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 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* (Limerick, Ireland) *(EASE '16)*. Association for Computing Machinery, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/2915970.2915984

[17] Fabio Ferreira and Marco Tulio Valente. 2023. Detecting code smells in React-based Web apps. *Information and Software Technology* 155 (2023), 107111.

[18] Martin Fowler. 2018. *Refactoring: improving the Design of Existing Code*. Addison-Wesley Professional.

[19] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1303–1313. https://doi.org/10.1145/3468264.3473924

[20] Shehzad Haider, Wajeeha Khalil, Ahmad Sami Al-Shamayleh, Adnan Akhunzada, and Abdullah Gani. 2023. Risk Factors and Practices for the Development of Open Source Software From Developers' Perspective. *IEEE Access* 11 (2023), 63333–63350. https://doi.org/10.1109/ACCESS.2023.3267048

[21] Rusen Halepmollasi and Ayse Tosun. 2024. Exploring the relationship between refactoring and code debt indicators. *Journal of Software: Evolution and Process* 36, 1 (2024), e2447.

[22] Zhewei Hu, Yang Song, and Edward F. Gehringer. 2019. A Test-Driven Approach to Improving Student Contributions to Open-Source Projects. In *2019 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE43999.2019.9028521

[23] Scientific Toolworks Inc. 2023. *Understand Software Metrics*. https://documentation.scitools.com/pdf/metricsdoc.pdf

[24] Jyun-Yu Jiang, Pu-Jen Cheng, and Wei Wang. 2017. Open Source Repository Recommendation in Social Coding. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Shinjuku, Tokyo, Japan) *(SIGIR '17)*. Association for Computing Machinery, New York, NY, USA, 1173–1176. https://doi.org/10.1145/3077136.3080753

[25] Yiqiao Jin, Yunsheng Bai, Yanqiao Zhu, Yizhou Sun, and Wei Wang. 2023. Code Recommendation for Open Source Software Developers. In *Proceedings of the ACM Web Conference 2023* (<conf-loc>, <city>Austin</city>, <state>TX</state>, <country>USA</country>, </conf-loc>) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 1324–1333. https://doi.org/10.1145/3543507.3583503

[26] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 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* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 119–125. https://doi.org/10.1145/3304221.3319780

[27] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A Tutoring System to Learn Code Refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 562–568. https://doi.org/10.1145/3408877.3432526

[28] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *2009 16th Working Conference on Reverse Engineering*. 75–84. https://doi.org/10.1109/WCRE.2009.28

[29] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610. https://doi.org/10.1016/j.jss.2020.110610

[30] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

[31] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Information and Software Technology* 83 (2017), 14–34. https://doi.org/10.1016/j.infsof.2016.11.009

[32] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 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* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 52–61. https://doi.org/10.1145/3422392.3422419

[33] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 2021. *How Do Code Smell Co-Occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective*. Association for Computing Machinery, New York, NY, USA, 54–63. https://doi.org/10.1145/3474624.3474642

[34] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* 4 (1976), 308–320. http://10.1109/TSE.1976.233837

[35] Nora McDonald and Sean Goggins. 2013. Performance and participation in open source software on GitHub. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems* (Paris, France) *(CHI EA '13)*. Association for Computing Machinery, New York, NY, USA, 139–144. https://doi.org/10.1145/2468356.2468382

[36] Aziz Nanthaamornphong and Ekkarat Boonchieng. 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 (Formerly: Recent Patents on Computer Science)* 16, 7 (2023), 20–33. https://doi.org/doi:10.2174/2666255816666230222112313

[37] Debora Maria Nascimento, Kenia Cox, Thiago Almeida, Wendell Sampaio, Roberto Almeida Bittencourt, Rodrigo Souza, and Christina Chavez. 2013. Using Open Source Projects in software engineering education: A systematic mapping study. In *2013 IEEE Frontiers in Education Conference (FIE)*. 1837–1843. https://doi.org/10.1109/FIE.2013.6685155

[38] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 440–451. https://doi.org/10.1145/2884781.2884868

[39] Gustavo Pinto, Clarice Ferreira, Cleice Souza, Igor Steinmacher, and Paulo Meirelles. 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)*. 147–157. https://doi.org/10.1109/ICSE-SEET.2019.00024

[40] Huilian Sophie Qiu, Anna Lieb, Jennifer Chou, Megan Carneal, Jasmine Mok, Emily Amspoker, Bogdan Vasilescu, and Laura Dabbish. 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* (<conf-loc>, <city>Hamburg</city>, <country>Germany</country>, </conf-loc>) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 552, 18 pages. https://doi.org/10.1145/3544548.3581317

[41] Fernanda Gomes Silva, Paulo Ezequiel D. Santos, and Christina von Flach. 2023. OSS in Software Engineering Education: Mapping Characteristics of Brazilian Instructors. *Journal of Software Engineering Research and Development* 11, 1 (Jan. 2023), 2:1 – 2:14. https://doi.org/10.5753/jserd.2023.1977

[42] Jefferson O. Silva, Igor Wiese, Daniel M. German, Christoph Treude, Marco A. Gerosa, and Igor Steinmacher. 2020. Google summer of code: Student motivations and contributions. *Journal of Systems and Software* 162 (2020), 110487. https://doi.org/10.1016/j.jss.2019.110487

[43] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, Carlos Lucena, and Rodrigo Paes. 2018. Identifying Design Problems in the Source Code: A Grounded Theory. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 921–931. https://doi.org/10.1145/3180155.3180239

[44] Diomidis Spinellis. 2021. Why computing students should contribute to open source software projects. *Commun. ACM* 64, 7 (jun 2021), 36–38. https://doi.org/10.1145/3437254

[45] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. 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* (Christchurch, New Zealand) *(EASE'18)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/3210459.3210466

[46] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[47] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[48] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218. https://doi.org/10.1016/j.infsof.2016.01.004