Automating Source Code Refactoring in the Classroom

Eman Abdullah AlOmar
Stevens Institute of Technology
Hoboken, United States
ealomar@stevens.edu

Mohamed Wiem Mkaouer
University of Michigan-Flint
Michigan, United States
mmkaouer@umich.edu

Ali Ouni
ETS Montreal, University of Quebec
Montreal, Quebec, Canada
ali.ouni@etsmtl.ca

ABSTRACT

Refactoring is the practice of improving software quality without altering its external behavior. Developers intuitively refactor their code for multiple purposes, such as improving program comprehension, reducing code complexity, dealing with technical debt, and removing code smells. However, no prior studies have exposed the students to an experience of the process of antipatterns *detection* and refactoring *correction*, and provided students with toolset to practice it. To understand and increase the awareness of refactoring concepts, in this paper, we aim to reflect on our experience with teaching refactoring and how it helps students become more aware of bad programming practices and the importance of correcting them via refactoring. This paper discusses the results of an experiment in the classroom that involved carrying out various refactoring activities for the purpose of removing antipatterns using JDeodorant, an IDE plugin that supports antipatterns detection and refactoring. The results of the quantitative and qualitative analysis with 171 students show that students tend to appreciate the idea of learning refactoring and are satisfied with various aspects of the JDeodorant plugin's operation. Through this experiment, refactoring can turn into a vital part of the computing educational plan. We envision our findings enabling educators to support students with refactoring tools tuned towards safer and trustworthy refactoring.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**.

KEYWORDS

refactoring, antipattern, quality, software engineering, education

ACM Reference Format:
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 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630787

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA

ACM ISBN 979-8-4007-0423-9/24/03…$15.00
https://doi.org/10.1145/3626252.3630787

1 INTRODUCTION

Design antipatterns are symptoms of poor choices at the software architecture level. These bad programming practices typically violate object-oriented design principles, such as *Single Responsibility* and *Law of Demeter*. The existence of these design antipatterns often leads to the degradation of software architectures, making them difficult to understand, reuse, and evolve. It is important to note that these antipatterns are different from coding errors, and do not directly lead to compiler or logical faults, but various studies have demonstrated how the existence of antipatterns makes the code significantly more prone to errors [11, 27, 28].

Two popular examples of design antipatterns are `God Class` and `Feature Envy`. The first characterizes classes that are abnormally large and monopolize most of the system's behavior by controlling a significant number of other coupled classes. The decomposition of such a class is critical to maintaining the modular design of the system. The second is related to methods that heavily rely on methods and attributes that are outside of its class more than those inside it. This is a symptom of a misplaced method that needs to be moved to a class to make it more cohesive.

To cope with these antipatterns, refactoring has emerged as a *de-facto* practice to improve software quality through the removal of antipatterns [16]. Refactoring is the art of improving source code internal design, without altering its external behavior [6, 18].

Several studies have proposed methods for teaching code refactoring through the identification of duplicate and dead code, and bad naming conventions [22, 23, 25, 26]. While these techniques play a role in improving the student's understanding of refactoring, it is critical to expose students to deeper design-level antipatterns that are frequently found even in well-engineered projects [30], and harder to fix [9]. For instance, early exposure to God Classes and Feature Envies would help reduce their prevalence in the future.

Therefore, the *goal* of this paper is to increase awareness of bad programming practices, *i.e.,* design-level antipatterns, and the importance of correcting them through the application of appropriate refactoring operations. Hence, we perform a series of assignments where students are asked to reason over how to refactor the `God Class` and `Feature Envy` design antipatterns. We chose these specific antipatterns on the basis of their frequent refactoring by developers in various systems[1].

We report our experience using JDeodorant [35], an integrated development environment (IDE) plugin, to support students in finding suitable refactorings. We chose JDeodorant because it is widely used by researchers as the state-of-the-art benchmark to assess the precision of refactoring techniques. JDeodorant is also widely adopted by practitioners to improve their systems' design.

[1]Based on tool usage statistics: https://users.encs.concordia.ca/ nikolaos/."

We adopted the reflective learning strategy when designing the refactoring assignments [12]. In fact, we follow Ash and Clayton's *DEAL* model [8] as we aim to let students first construct and *describe* an initial refactoring solution before *examining* other candidate solutions recommended by JDeodorant, to finally *articulate* on the difference between their solution and the ones recommended by the tool. We executed these assignments in undergraduate and graduate software engineering courses at Stevens Institute of Technology and Rochester Institute of Technology. We analyzed 171 student refactoring submissions in terms of two dimensions. The first dimension is empirical, as we assess the quality of students' refactored code in contrast with JDeodorant's, to extract any knowledge gap. This dimension's outcome reveals how `God class` antipattern tends to be harder for students to refactor, compared to `Feature Envy`, and how the use of JDeodorant has facilitated the correction of these hard instances. The second dimension is qualitative, where we survey students to sense their feedback on the tool's usefulness, usability, and functionality. The results of the survey show that the vast majority of students (87% responses) found the plugin to be useful, and usable, and were satisfied with its operation. Finally, we reflect on the importance of reinforcing software design principles and patterns. Therefore, we foresee students' usage of JDeodorant as an opportunity to raise awareness of the detection of antipatterns and their correction measures.

This paper contributes to the broad adoption of refactoring by (i) designing practical assignments that first challenge students' abilities to refactor design-level problems, then second provide them with candidate solutions to reason over and choose based on their potential impact on quality, and (ii) reporting the experience of using the JDeodorant plugin. This experiment enabled instructors to design personalized, hands-on assignments and support students in learning how to use refactoring features in the IDE. It also achieves another learning objective, since a recent study has shown that developers rarely use the built-in IDE features when refactoring their code, increasing the error-proneness of their changes [19]. As part of this paper's contributions, we provide the assignment description, the tool documentation, and statistical tests for educators to replicate and extend [1].

2 JDEODORANT WORKFLOW

JDeodorant [35] stands as one of the popular refactoring tools that have been provided as an Eclipse and IntelliJ IDEA plugins. It automatically detects antipatterns, including `Feature Envy`, `State Checking`, `Type Checking`, `Long Method`, `God Class`, `Duplicate Code`, and `Refused Bequest`, and for each detected antipattern, it offers its correction by providing a list of candidate refactorings that developers have to choose from. Developers, are then responsible for choosing the most adequate refactoring operations according to their design choices and preferences.

To illustrate the workflow of JDeodorant, we choose to fix the `Feature Envy` antipattern that may exist in the Gantt[2] project. We open the Gantt project using the Eclipse IDE, with JDeodorant already installed as a plugin. Then, we enable the plugin by clicking on `Bad Smells` in the menu bar (Step 1), and then we click on `Feature Envy`. To identify all instances of this antipattern, the plugin internally parses all the project methods to generate their corresponding

[2]https://github.com/bardsoftware/ganttproject

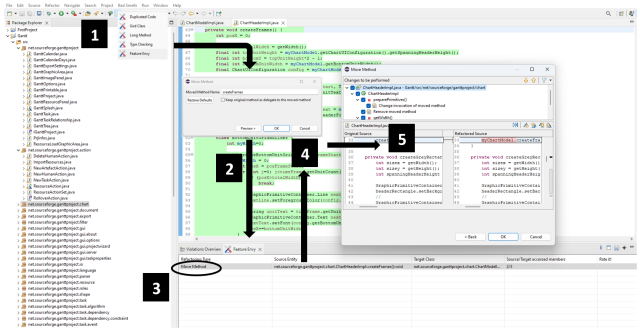


Figure 1: JDeodorant workflow.

Abstract Syntax Tree (AST) representations, which in turn allows the tool to determine whether a method matches the pattern of a `Feature Envy` antipattern. Once the detection process is done, all `Feature Envy` instances are shown in the plugin view (Step 2), along with their corresponding refactoring suggestions (Step 3). For example, as can be seen in Figure 1, the `createFrame` method, with 55 lines of code, and located in the class `CharHeaderImpl`, is flagged as a `Feature Envy`. The plugin also proposed a couple of candidate `Move Method` refactorings for us to choose from. If we choose to fix the `creatFrame` method by selecting the first refactoring suggestion, the plugin displays the method being refactored (Step 4), and internally calls the `Move Method` built-in feature inside Eclipse. This feature initiates the refactoring process by opening a preview window (Step 5). This window shows the original code of the `createFrame` method, side by side with a preview of refactored code, which would show the result of moving `createFrame` to another class. If we confirm the refactoring, it will automatically be applied to the source code.

3 EXPERIMENTAL SETUP

3.1 Teaching Context and Participants

The study is performed in undergraduate and graduate courses taught at Stevens Institute of Technology and Rochester Institute of Technology. The courses cover various concepts related to software analysis and testing, along with practical tools, widely used in the open-source community. Students were also given several hands-on assignments in topics including software quality metrics, code refactoring, bug management, unit and mutation testing, and technical debt. Before conducting the experiment, students acquired the necessary background by learning the following concepts: (1) code quality (teaching quality and how to measure it), (2) design antipatterns (teaching violations of design principles, and their detections rules), and (3) code refactoring (teaching refactoring recipes and operations). The experiment's assignments constituted 7.5% of the final grade. It was due 14 days each after the concepts were taught.

3.2 Assignments Content and Format

We adopt Ash and Clayton's reflection model by gathering evidence (refactored code) that can be examined to identify any gaps in the

state of refactoring practice (inability to correct certain antipatterns), with the intent to improve it (provide alternative correction mediums).

Initially, students are asked to analyze one version of a Java software system of their choice approved by the instructor to ensure its eligibility based on popularity, besides making sure it correctly compiles since JDeodorant requires it. The rationale behind giving students the choice of project, is to let them choose one that they are comfortable with, and fits into their interests. For students who do not want to search for a project, they are given a list that the instructor has already curated. We selected these projects [1] because they contain the antipatterns that we are interested in. We conduct our experiments through two assignments: In the first assignment, students are asked to fix the two antipatterns (*i.e.,* `God Class` and `Feature Envy`) and provide a sequence of refactoring operations that will fix multiple instances of these antipatterns. The submissions to this assignment constitute the *Manual Refactoring*. In the second assignment, students are asked to set up and run JDeodorant to analyze the chosen project production code. Upon running JDeodorant, students are required to choose at least 2 antipatterns instances from 2 different antipatterns types supported by the tool (4 in total), and then analyze JDeodorant recommendations to choose the potential refactoring operations to fix them. Since JDeodorant gives many recommendations on how to fix the same antipatterns instance, based on the students' understanding of problems' symptoms, they would need to reason when choosing the code changes that remove the smells while fitting properly in the system's design. The submissions to this assignment constitute the *Assisted Refactoring*. In summary, the students followed these steps:

(1) Manually fix the detected antipatterns types: (*i*) `God Class`, and (*ii*) `Feature Envy`.
(2) Provide a sequence of refactoring operations that will fix multiple instances of those antipatterns (*i.e., Manual Refactoring*).
(3) Justify the choices regarding refactoring decisions for each fixed antipattern type.
(4) Install the Eclipse or IntelliJ IDEA plug-in for JDeodorant.
(5) Run JDeodorant on a project of students' choice and select 2 instances of each of the 2 following antipatterns types: (*i*) `God Class`, and (*ii*) `Feature Envy`.
(6) Look at the refactoring recommendations by JDeodorant, and choose which ones to be executed. Students keep refactoring until processing all their chosen smell instances.
(7) Report the findings: chosen antipattern instances, chosen refactoring operations and results (*i.e., Assisted Refactoring*).
(8) Add to the report a concise comment about the experience with JDeodorant (Optional).

Students were evaluated based on two aspects, (1) concept understanding: assessment of students' ability to apply the right refactoring to fix antipatterns; and (2) program analysis: assessment of whether students are able to execute refactoring and verify the preserved behavior. Students were not evaluated on their perception of the code, to avoid any cognitive bias that may occur under the pressure of being graded. Also, we anonymized the feedback, and made it optional, to only collect information from students who were serious about it, which will increase the magnitude of provided experience. Despite it being not mandatory, the majority of students (96.07%) chose to complete it.

The assignment was performed over four consecutive semesters. 171 students, primarily from computer science (CS) and software engineering (SE) majors, were enrolled during these semesters, and completed assignments.

3.3 Data Analysis

We analyzed the responses to open-ended questions to create a comprehensive high-level list of themes by adopting a thematic analysis approach based on guidelines provided by Cruzes *et al.* [15]. Thematic analysis is one of the most used methods in Software Engineering literature [4, 5, 32]. This is a technique for identifying and recording patterns (or "themes") within a collection of descriptive labels, which we call "codes". For each response, we proceeded with the analysis using the following steps: *i*) Initial reading of the survey responses; *ii*) Generating initial codes (*i.e.,* labels) for each response; *iii*) Translating codes into themes, sub-themes, and higher-order themes; *iv*) Reviewing the themes to find opportunities for merging; *v*) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence. The above-mentioned steps were performed independently by two authors. One author performed the labeling of students' comments independently from the other author who was responsible for reviewing the currently drafted themes. At the end of each iteration, the authors met and refined the themes.

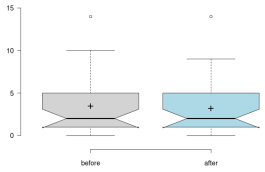
4 RESULTS

4.1 Quantitative Analysis

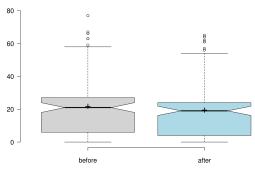
Table 1: Statistical test.

Antipattern	Approach	Impact	*p*-value	Cliff's delta (δ)
`God Class`	Manual	+ve	***0.01***	0.05 (Negligible)
	Assisted	+ve	***1.89e-161***	0.76 (Large)
`Feature Envy`	Manual	+ve	***2.22e-06***	0.19 (Small)
	Assisted	+ve	***1.46e-161***	0.76 (Large)

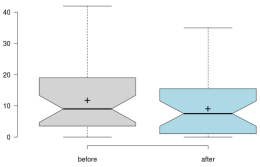
To show the effectiveness of JDeodorant in educating students about better-making design decisions, we count the number of `God Class`, and `Feature Envy` antipatterns before and after refactoring, initially when students refactored the antipatterns manually (referred to as *Manual Refactoring*), and then when students refactored the antipatterns based on JDeodorant recommendations (referred to as *Assisted Refactoring*). Figure 2 reports the box plots depicting the distribution of each group value, clustered by the two abovementioned antipatterns. We used the Wilcoxon test [36] to test the significance of the difference between the group's values. This non-parametric test checks continuous or ordinal data for a significant difference between two dependent groups. Our hypothesis is formulated to test, for each antipattern and for each refactoring (*manual* or *assisted*), whether the number of instances of the antipattern in the code after the refactoring is significantly lower than the number



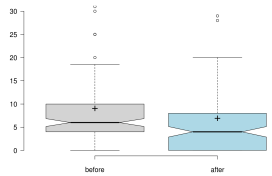
(a) Impact of Manual Refactoring on God Class Distribution



(b) Impact of Assisted Refactoring on God Class Distribution



(c) Impact of Manual Refactoring on Feature Envy Distribution



(d) Impact of Assisted Refactoring on Feature Envy Distribution

Figure 2: Boxplots of `God Class`, and `Feature Envy` antipattern instances addressed by students.

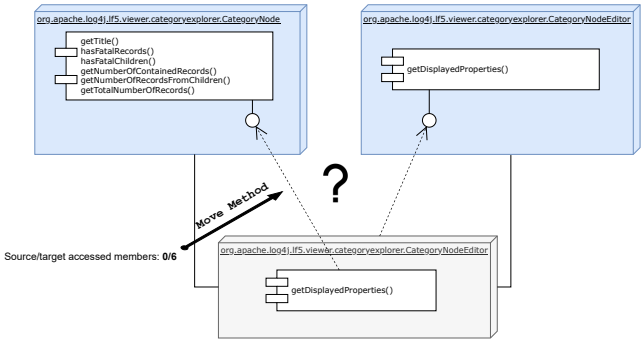


Figure 3: `Feature Envy` example from the Log4J project [2].

of instances of the antipatterns before the refactoring. The difference is considered statistically significant if the p-value is less than 0.05. Furthermore, we used Cliff's delta (δ) effect size to estimate the magnitude of the differences. Regarding its interpretation, we follow the guidelines reported by [20]: Negligible for $|\delta| < 0.147$; Small for $0.147 \leq |\delta| < 0.33$; Medium for $0.33 \leq |\delta| < 0.474$; and Large for $|\delta| \geq 0.474$.

The count of corrected antipatterns was done manually by the authors. Since the complete list of students' submissions is too large to examine manually, we selected a statistically significant sample for our analysis and annotated 62 submissions. This quantity roughly equates to a sample size with a confidence level of 95%, with a margin of error of 10%. As can be seen in Figure 2, antipatterns (*i.e.,* `God Class` and `Feature Envy`) before refactoring are larger than the antipatterns after refactoring. While the difference was statistically significant (0.01 and 2.22e-06 for `God Class` and `Feature Envy`, respectively), the magnitude of the difference is negligible and small for `God Class` and `Feature Envy`, respectively. We conjecture that although there is quality improvement as the number of antipatterns decreased, there were many instances where students' *Manual Refactorings* could not remove certain antipatterns, particularly the God Classes instances. However, *assisted refactoring* also had a positive impact on quality, since the number of antipatterns (*i.e.,* `God Class` and `Feature Envy`) before refactoring is greater than the antipatterns after refactoring (1.89e-161 and 1.46e-161 for `God Class` and `Feature Envy`, respectively). However, the main difference lies in the magnitude of the difference (Cliff Delta), which was large for the *Assisted Refactoring*, according to Table 1. We conclude that JDeodorant's assistance was beneficial in improving students' design decisions. Figure 3 shows an example of a `Feature Envy` design antipattern from one of the student's submissions. The `CategoryNodeEditor` class has a method called `getDisplayedProperties` that seems overly interested in the properties of the `CategoryNode` class. The method `getDisplayedProperties` calls many methods from `CategoryNode` class more than its own class methods (Coupling = 6). This indicates that the method should instead belong to the `CategoryNode` class. When the student applied the `Move Method` refactoring, as recommended by JDeodorant, the `Feature Envy` antipattern was removed, along with a decrease in the system's overall coupling.

Observation. ***Not all antipatterns are easily refactorable.*** It is important to note that despite students' efforts to remove antipatterns, our analysis of *Manual Refactoring* code shows how many instances of antipatterns existed after the refactoring session. In particular, God Classes are difficult to refactor, as the magnitude of the difference is negligible in Table 1. Previous research has demonstrated how God classes tend to be hard to fix in industry [7]. So, we emphasize the importance of raising students' awareness, as a preventive measure, to avoid creating God classes.

Observation. ***Understanding the impact of refactoring on quality is challenging for students.*** Although both refactoring sessions aim to improve quality, students realize, when they compare their manual refactoring with the assisted one, that not all refactorings can be equally beneficial to design quality. For instance, the process of extracting multiple classes, to remove a *God Class* antipattern, will eventually increase the number of classes per package, which is also considered an increase in system complexity according to the CK quality metrics [13]. Thus, students should

Table 2: Student's insight about the usefulness, usability and functionality of the tool.

Theme	Sub-theme	Example (Excerpts from a related student's comment)
Usefulness	Efficiency	"*JDeodorant is undoubtedly a very convenient tool for developers, he can easily fix various code smells and* ***improve the development efficiency*** *of developers.*"
	Quality	"*It can help to prioritize refactoring instances by selecting them based on their number of methods and attributes they have. It also gives more details about classes and have an* ***insight of the impact of refactoring***."
	Automation	"*Overall my experience with the plugin was good. It takes most of the refactoring work and* ***basically automates it***."
	Awareness	"*The product seemed to work well. Additionally, its ability to tell what refactoring to do and actually perform it seems super helpful, and* ***more helpful if I actually knew what I was doing when refactoring***."
	Experience	"*My experience with JDeodorant was quite pleasant. It* ***simplified the process*** *of addressing flaws in the code and* ***for someone with limited software engineering experience*** *related to code smells it was very informative.*"
Usability	Graphical design	"*the very intuitive* ***graphical design*** *of the plugin (for instance, doing things such as highlighting code that it changes in green)* ***makes such reviews far simpler*** *than alternative approaches may provide.*"
	Preview	"*I particularly liked IntelliJ's display* ***showing side-by-side diffs of what changed****; it made it clear what was changing, which in turn made it easier to interpret why the tool was suggesting the change.*"
	Visualization	"*The JDeodorant version for eclipse* ***provides the visual context/flow chart*** *containing the breakdown of the bad smells and the highlighted code. That proved to be very beneficial during the refactoring process, especially with god class.*"
	Documentation	"*It is also an automated process, which is User friendly and* ***it gives guidelines for the usage*** *and it also pre-evaluates the refactoring.*"
Functionality	Design antipatterns	"*I found that jDeodorant did a* ***great job at detecting any issues*** *that were found.*"
	Refactoring	"*I am happy I picked a smaller project to work on, it is significantly easier to see the effects of the changes. Most of the* ***refactoring that was suggested were various kinds of extraction****, mostly methods.*"
Recommendation	Quality	"*It would have been nicer if the project* ***checked for refactorings*** *again automatically* ***after applying*** *one instead of the user needing to use it.*"
	Design antipatterns	"*I am really impressed with how this tool can detect the code smells and suggest refactoring to solve the problems. However, [...], it can* ***only detect three types of code smells****. Many software has a wide range of other code smells not only (long method, god class, feature envy) which are not possible to detect using this tool. "*
	Refactoring	"*some of it's* ***suggestion of the refactoring is incompleted****, the suggestion name and logic detection has missed some part of the code. So I feel this tool can only use as an quick reference in the code quality review*"
	Testing	"*Its* ***hard to find*** *if refactoring did* ***break any of the feature*** *or functionality remains the same even after refactoring.*"

consider these trade-offs as they make their choices. In this context, quality gate tools, such as Understand[3] and SonarQube[4] can be deployed to measure the quality before and after the application of refactoring. This might strengthen students' understanding of writing well-structured code and raise their confidence to perform the recommended refactoring.

4.2 Qualitative Analysis

In Table 2, we report the main thoughts, comments, and suggestions about the overall impression of the usefulness, usability, functionality, and recommendation of the tool, according to the conducted labeling. Figure 4 shows the percentages of students' insight. As can be seen, the 'Functionality' and 'Usefulness' categories had the highest number of responses, with a response ratio of 34.6% and 32.1%, respectively. The category 'Recommendation' was the third most popular category with 21.8%, followed by 'Usability', which had a ratio of 11.5%. This finding indicates what students mainly care about when using the tool. Table 2 presents samples of the students' comments to illustrate their impression of each theme.

Usefulness. Generally, the respondents found the plugin useful in terms of five main aspects: efficiency, quality, automation, awareness, and experience. 40 out of 171 students commented that JDeodorant is very intuitive to use and was quite efficient to find refactoring opportunities, and convenient for developers who would not have to examine and refactor antipatterns manually. 30 students communicated that eliminating antipatterns assists in increasing its *readability* while reducing its *coupling* and *complexity*, which helps improve overall code quality. A few students revealed that the tool's ability to identify antipatterns within a selected file allows them to only correct errors in a specific location/file of their interest, instead of inspecting the entire project. Further, two students commented that the tool aids less experienced developers in identifying the antipatterns when updating a source file that they are not necessarily familiar with. Additionally, a group of students mentioned that the tool helps less experienced and novice developers in writing well-structured code.

Usability. Based on the feedback provided by the students, the key areas of usability related to the graphical design, preview, visualization and documentation. Five students pointed out that the graphical design of the plugin is intuitive, especially the IDE's display feature showing side-by-side diffs of what changed. This preview feature makes it clear what was changing and why the tool was suggesting the change. Other comments also stated the importance of the preview feature, which allows users to foresee the impact of the change without actually performing it. Two students reported the useful feature of antipattern visualization as a flow chart, as it allows locating the *hot* areas in code that encapsulate a large set of smells. Lastly, the documentation of the plugin is written and easy to follow.

Functionality. According to the students' feedback about the tool's functional features, 34.6% of the students' comments show a couple of appreciation for the supported antipatterns by the tool, and how this feature helps in better understanding the concepts in a real-world scenario. Additionally, the students commented on their ability to practice a variety type of refactoring operations according to their removal of the antipatterns.

[3]https://scitools.com/
[4]https://sonarqube.org

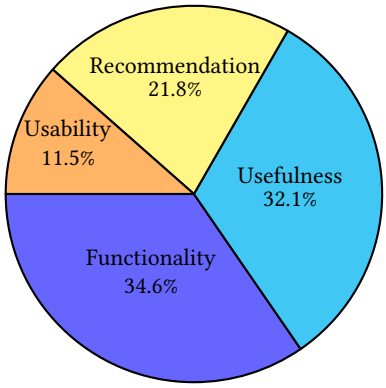


Figure 4: Students' perception about the refactoring tool.

Recommendation. From the students' feedback, we have also extracted suggestions to improve the tool's features. 21.8% of the students' comments show a couple of suggested changes as a recommendation or refactoring support to be made to the tool's operation or UI. We found that the students pointed out some of the recommendations related to quality, antipatterns, refactoring, and testing. Students recommend the tool to perform a validation after performing the refactoring application, support more antipatterns, perform the complete application of refactoring, and perform testing to check the behavior preservation of code transformation after refactoring.

5 REFLECTIONS

📌 **Reflection #1:** ***Raising the awareness of antipatterns and their refactoring strategies.*** Previous studies [3, 24, 29, 33, 34] have proposed different methods to teach refactoring, but did not provide students with toolset to practice it. It is still a manual process that might contribute in a limited way in the long run. However, if students learn how to automate refactoring, it helps them to better refactor the code while preserving the behavior and making the code less prone to error. Further, in education, there are many concepts related to design principles that are taught in the classroom, such as SOLID and GRASP. However, the responses that we received from students, have shown the importance of covering design anti-patterns, and bad programming practices avoidance in curricula, as these topics are generally less popular, compared to design patterns. In many CS and SE curricula, instructors highlight software quality when teaching good programming practices and design patterns. This assignment complements it by revealing how deviation from best practices can lead to poor design choices that negatively impact the source code. Similarly to teaching blueprints of design patterns, students should also be exposed early to quality concerns and encouraged to improve the design of their own code. Moreover, one of the noteworthy points is that using the tool in the assignment helps less experienced and novice coders to write well-structured code. To further raise awareness, educators can include empirical evidence to enhance students' understanding of refactoring and antipatterns concepts and so students providing low-quality code can be convinced that these concepts help improve the quality of software systems.

📌 **Reflection #2:** ***Reinforcing software engineering principles and good development practices.*** Studies have shown how group-based project artifacts tend to be purposely over-engineered complex models as a means to showcase the ability to design complex systems [10, 14]. Through this assignment, students would realize that over-engineered systems tend to contain more design anti-patterns, and therefore simplifying them is necessary for their code to become easier to maintain in the future. According to the students' insights, we believe that instructors need to highlight the desirable properties of refactoring tools (*e.g.,* quality improvement, developer perception, automated testing, etc.). Future educators and researchers are encouraged to revisit existing refactoring tools so that students can gain more confidence in using them. Furthermore, since the classical definition of refactoring focuses on preserving the behavior of the applied transformation, instructors may consider pointing out some behavior preservation strategies (*e.g.,* [6]) and explore their potential in assessing the correctness of the refactored code (*e.g.,* in the context of JDeodorant, students can run unit tests to verify that the applied refactoring was behaviorally preserved).

6 RELATED WORK

Smith *et al.* and Stoecklin *et al.* [33, 34] introduced an incremental approach by focusing on lessons from an innovative pedagogical approach to teaching refactoring, such as self-documenting code and better recognizing code. The authors conclude that refactoring can become an integral component of the computer science curriculum by reinforcing software engineering principles. Rabb [31] introduced CodeSmellExplorer to familiarize users with good coding practices by visualizing an interactive graph network of antipatterns and connected refactorings. Lobez *et al.* [29] described e-activities for teaching refactoring by following Bloom's taxonomy (*i.e.,* proposing activities to help with understanding a concept, applying refactorings in the context of and synthesizing of the use of refactorings in open source projects). Elezi *et al.* [17] proposed a gamification system that tracks and rewards refactorings during development. Haendler and Neumann [21] explored the challenges of designing serious games for refactoring on real-world code artifacts. Specifically, they proposed a game design where students can compete either against a predefined benchmark (technical debt) or against each other. In a follow-up work, Haendler *et al.* [22, 23] developed an interactive tutoring system for training software refactoring. Keuning *et al.* [25, 26] to teach students to refactor functionally correct code. More recently, Izu *et al.* [24] proposed a lab-based resource to help novices identify and refactor antipatterns when writing conditional statements.

7 CONCLUSION AND FUTURE WORK

In this study, we demonstrate the use of the JDeodorant tool to support students with refactoring antipatterns. Overall, the participants rated various aspects of the plugin highly, while also providing valuable ideas for future development. We envision our findings enabling educators to support students with refactoring tools tuned towards safer refactoring. Future work in this area includes investigating students' understanding of refactoring using various real-world applications in a semester-long course project.

REFERENCES

[1] [n. d.]. https://refactorings.github.io/education/.

[2] [n. d.]. https://github.com/apache/logging-log4j2.

[3] Shamsa Abid, Hamid Abdul Basit, and Naveed Arshad. 2015. Reflections on teaching refactoring: A tale of two projects. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 225–230.

[4] Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. 2023. On the use of static analysis to engage students with software quality improvement: An experience with pmd. *arXiv preprint arXiv:2302.05554* (2023).

[5] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack. (2022), 1–13.

[6] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* (2021), 106675.

[7] Nicolas Anquetil, Anne Etien, Gaelle Andreo, and Stéphane Ducasse. 2019. Decomposing god classes at siemens. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 169–180.

[8] Sarah L Ash and Patti H Clayton. 2009. Generating, deepening, and documenting learning: The power of critical reflection in applied learning. (2009).

[9] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.

[10] Leema K Berland, Taylor H Martin, Pat Ko, Stephanie Baker Peacock, Jennifer J Rudolph, and Chris Golubski. 2013. Student learning in challenge-based engineering curricula. *Journal of Pre-College Engineering Education Research (J-PEER)* 3, 1 (2013), 5.

[11] Narjes Bessghaier, Ali Ouni, and Mohamed Wiem Mkaouer. 2021. A longitudinal exploratory study on code smells in server side web applications. *Software Quality Journal* 29, 4 (2021), 901–941.

[12] Anne Brockbank and Ian McGill. 2007. *Facilitating reflective learning in higher education*. McGraw-Hill Education (UK).

[13] 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.

[14] Carol L Colbeck, Susan E Campbell, and Stefani A Bjorklund. 2000. Grouping in the dark: What college students learn from group projects. *The Journal of Higher Education* 71, 1 (2000), 60–83.

[15] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. IEEE, 275–284.

[16] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1992), 29–30.

[17] Leonard Elezi, Sara Sali, Serge Demeyer, Alessandro Murgia, and Javier Pérez. 2016. A game of refactoring: Studying the impact of gamification in software refactoring. In *Proceedings of the Scientific Workshop Proceedings of XP2016*. 1–6.

[18] Martin Fowler, Kent Beck, John Brant, William Opdyke, and don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. http://dl.acm.org/citation.cfm?id=311424

[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*. 1303–1313.

[20] Robert J Grissom and John J Kim. 2005. *Effect sizes for research : a broad practical approach*. Mahwah, N.J. ; London : Lawrence Erlbaum Associates. Formerly CIP.

[21] Thorsten Haendler and Gustaf Neumann. 2019. Serious refactoring games. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.

[22] Thorsten Haendler, Gustaf Neumann, and Fiodor Smirnov. 2019. An interactive tutoring system for training software refactoring. *Instructor* 1 (2019), 4.

[23] Thorsten Haendler, Gustaf Neumann, and Fiodor Smirnov. 2019. RefacTutor: an interactive tutoring system for software refactoring. In *International Conference on Computer Supported Education*. Springer, 236–261.

[24] Cruz Izu, Paul Denny, and Sayoni Roy. 2022. A Resource to Support Novices Refactoring Conditional Statements. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. 344–350.

[25] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.

[26] 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*. 562–568.

[27] 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*. IEEE, 75–84.

[28] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[29] Carlos López, Jesús M Alonso, Raúl Marticorena, and Jesús M Maudes. 2014. Design of e-activities for the learning of code refactoring tasks. In *2014 International Symposium on Computers in Education (SIIE)*. IEEE, 35–40.

[30] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[31] Felix Raab. 2012. CodeSmellExplorer: Tangible exploration of code smells and refactorings. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 261–262.

[32] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[33] Suzanne Smith, Sara Stoecklin, and Catharina Serino. 2006. An innovative approach to teaching refactoring. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 349–353.

[34] Sara Stoecklin, Suzanne Smith, and Catharina Serino. 2007. Teaching students to build well formed object-oriented methods through refactoring. *ACM SIGCSE Bulletin* 39, 1 (2007), 145–149.

[35] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 4–14.

[36] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.