Expanding Code Assessment: A Qualitative Feedback System for Beginning Students

Raimundo Meliana de Carvalho Filho, Carlos de Salles Soares Neto and Davi Viana dos Santos
Departamento de Informática, Universidade Federal do Maranhão, Brazil

Keywords: Qualitative Feedback, Code Quality, Programming Education.

Abstract: Although online judge systems are effective in verifying code correctness, they tend to provide only binary answers related to code functionality, limiting students' understanding of errors and opportunities for improvement. The development of qualitative feedback system that provides students with more comprehensive recommendations and guidance is promising as it fills this gap. That said, this study investigates the extent to which novice programming students consider and incorporate feedback suggestions into their coding practices, with the aim of evaluating the ability of a feedback system to influence and improve the quality of programming learners' code. To achieve this goal, we examined the correlation between certain aspects of code quality raised by the system and the frequency of resubmissions. The results revealed a correlation between resubmissions to the system and the presence of code smells related to the naming of variables in the students' codes ($r = -0.4718$, $p < 0.05$). These findings reinforce the importance of code quality feedback and highlight the need for code quality features in online judging environments.

1 INTRODUCTION

In the context of programming education, Online Judge systems represent and play an important role in student assessment and feedback (Zhou et al., 2018). These systems provide a virtual environment in which students can submit their code and obtain automatic results based on test cases (Wasik et al., 2018).

Although Online Judges are effective in verifying the correctness of code, their quantitative approach generally does not provide detailed information about the qualitative aspects of code (Complexity, Variable naming, Refactoring) produced by students. Assessment solely focused on binary answers (yes or no) can limit students' understanding of mistakes made and opportunities for improvement in their programs.

In view of the above, motivation arises for the development of qualitative feedback to student learners. The idea is to provide recommendations and guidance that go beyond simple verification of correctness, presenting subjective and qualitative aspects of programming.

The research question guiding this work is: To what extent do novice student programmers incorporate and apply feedback suggestions into their coding practices? The research aims to evaluate the ability of a qualitative feedback system to influence and improve code quality in this group.

For the study, a qualitative feedback system focused on three aspects of code quality that we consider relevant for beginner students was developed: Variable names, code complexity and refactoring. Feedback related to variable names is the main differentiator of this system, since there are few approaches in this regard.

A correlation analysis between the types of code smells (complexity, variable names, refactoring) and student resubmissions revealed a significant correlation between variable names and the number of resubmissions. This suggests that the quality of variable naming improves with qualitative feedback and code refactoring. These results indicate that the qualitative feedback system can improve the quality of beginning students' code, especially regarding variable names.

Section 2 presents related work, contextualizing the novel aspects and advantages of the present work. In Section 3, we detail the tool developed, its purpose and its main functionalities and characteristics. Then, in Section 4, we present the evaluation of the tool regarding the research objectives, including the description of the participants, data collection and

Filho, R., Neto, C. and Santos, D.
Expanding Code Assessment: A Qualitative Feedback System for Beginning Students.
DOI: 10.5220/0012741600003693
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Conference on Computer Supported Education (CSEDU 2024) - Volume 1*, pages 578-584
ISBN: 978-989-758-697-2; ISSN: 2184-5026
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

methods for data analysis. In Sections 5 and 6, we present the results and conclusions of the work.

2 RELATED WORK

Online Judge have become increasingly prevalent, both in programming competitions and in the academic and educational setting. In the educational context, these systems play the role of tutors, providing a quick assessment of the accuracy of submitted codes. However, the assessment conducted by these environments often do not fully capture the complexities present in assessment made by teachers (Zhou et al., 2018).

Code assessment by a teacher goes beyond merely checking correctness, covering qualitative aspects and specific nuances. However, few comprehensive studies have been conducted related to qualitative aspects and feedback in Online Judge environments (Liu & Woo, 2020).

The challenge lies in the need to develop and improve these systems to incorporate a more comprehensive assessment, which goes beyond error detection and considers the intrinsic quality of the code. Research in this field is crucial to ensuring that Online Judges can offer support and assessment as thorough as that provided by experienced teachers, thus promoting a richer and more effective educational environment.

With this in mind, Araujo et al. (2016) propose a set of measures to capture code quality and generate useful feedback for novice programmers. The proposed measures are based on traditional software quality metrics and can be obtained automatically, as long as there is a reference solution. The research discusses the qualitative aspects of code that instructors typically evaluate in programming assignments. The set of proposed measures is evaluated through a case study and an experiment. The results show that the use of software metrics can improve the feedback provided to students and instructors.

In the work presented by Urell and Wallace (2019), the authors show WebTA, a feedback system focused on programming style anti-patterns in the early stages and with different forms of identification and automatic treatment of these anti-patterns. WebTA takes the promising parts of student submissions and suggests more meaningful fixes than typical compiler error messages. The study highlights the importance of providing tailored automated feedback to novice programmers, addressing programming anti-patterns at early stages.

Keuning et al. (2020) present a tutoring system for programming that focuses on teaching students how to refactor functionally correct code, with an emphasis on method-level refactorings, control flow expressions, and language structures. The system provides automated feedback and layered tips to support students in their learning process. The study of 133 students using the system provides insights into how students approach exercises and how they use feedback and tips to refactor code.

Orr’s (2020) work presents a rule-based system aimed at evaluating student programs for design quality and providing personalized, accurate feedback to assist in their improvement. By implementing this online system, students were empowered to receive quick feedback and make necessary refinements before submitting their assignments. The study’s findings revealed a significant reduction in design quality flaws across multiple assignments, highlighting the system’s positive impact on improving the overall quality of student submissions.

In the work of Jiang et al. (2020), the authors present CompareCFG, a system that provides automated visual feedback on code quality through control flow graphs (CFGs). CompareCFG generates visualizations of students’ submissions and allows them to compare their own code with less complex submissions. The system also provides actionable feedback by identifying specific issues that can reduce code complexity. The tool was used by 5 software engineering students to evaluate its usability and the impact generated by the feedback on the students. Statistical analyzes from the pilot study show that CompareCFG provides useful feedback and helps students improve code quality and complexity.

Birillo et al. (2022) present Hyperstyle, a code analysis tool that evaluates the quality of programming solutions in the educational context. Hyperstyle focused on various categories of code quality, as it aggregated several professional tools. The study compares Hyperstyle with another educational code analysis tool (Tutor) in terms of number of errors before and after using the tools, showing that Hyperstyle results in an improvement in the quality of the code submitted by students. The tool also proves to be useful an impactful in the educational context, offering support for several programming languages.

In a study by Liu and Woo (2020), an Online Judge system was developed that evaluates not only the correctness, but also the quality of the code submitted by students. The quality detection module was developed using SonarQube [1]functionalities and tested on a set of 2.000 python codes from undergraduate students to verify its practical usefulness. As a result, the study presents the most frequent errors, commonly made by beginners, obtained using the tool.

Research stands out for presenting a qualitative feedback system, focused on offering suggestions related to practices for choosing variable names and managing code complexity and refactoring. Unlike conventional approaches, which often focus only on detecting errors, our system seeks to increase the intrinsic quality of the code, promoting good programming practices. Although other works address aspects such as correctness, general design, readability, maintainability and efficiency of the code, the present work differentiates itself by bringing suggestions related to the choice of variable names, a challenge faced and emphasized by other works (Chren et al., 2022; Orr, 2020). In general, the focus is on identifying code smells associated with the need for refactoring, improved suggestions for more descriptive variable names, and strategies for managing complexity, considering nuances that often escape automatic detection. Code smells refer to symptoms or indications that the source code may be poorly structured or in need of refactoring Fowler (1999).

This differentiated approach reinforces the importance of a comprehensive assessment that goes beyond traditional parameters, thus contributing to the training of more skilled programmers who are aware of the quality of their codes.

3 COSMOMENTOR

Cosmo is an internal educational and multitasking platform from Universidade Federal do Maranhão, developed and maintained by the Telemidia laboratory, dedicated to programming exercises for an introduction to Algorithms course. Its characteristics for solving programming activities are similar to traditional Online Judges. For a given activity, a user submits an algorithm as a solution and receives an answer about the functional correctness of that submission based on a battery of test case.

To complement the approach already offered by the Cosmo environment with qualitative feedback suggestions related to the students' codes, CosmoMentor was developed, which was developed in Python language as a Web Service and integrated into Cosmo environment.

[1] https://www.sonarsource.com

CosmoMentor is a tool for code quality analysis and feedback. It provides suggestions related to code quality aspects and is specifically designed to capture code quality issues from beginner programming students.

Within its functionalities, CosmoMentor enables the detection of code smells related to variable naming, complexity and refactoring and, thus, suggesting refactoring alternatives to the student in the form of suggestions. Its approach involves both static code analysis and natural language processing.

3.1 Static Analysis

For the static analysis, some of the functionalities present in the *pylint*, *radon* and *mccabe* libraries were used. The *pylint* library was used to detect deviations from python coding standards. *Pylint* was used to detect functional but unusual code snippets that often have a more common and accepted alternative in the context of python. Among these deviations, chaining comparison, variable swap, use of enumerate, number of Boolean expressions on the same line and line length.

The *radon* and *mccabe* libraries were used to measure the number of lines, Halstead metrics and cyclomatic complexity, respectively in both the students' codes and in a reference code for each question. A reference code is provided so that it is possible to compare the student code metrics with the reference ones and then generate suggestions for the students, similar to the approach found in the work of Araujo et al. (2016).

3.2 Variable Analysis

To improve the generation of automated feedback on variable naming, employ natural language processing techniques. The objective of this approach is to evaluate the relevance of variable names in the student's code, highlighting inappropriate names and proposing more appropriate alternatives. This is done by involving both an analysis of the variables present in the student's code in relation to a reference set, formed by the combination of both variables extracted from the question statement and a set of pre-defined reference variables.

Initially, a set of variables is extracted with natural language processing from the question statement text. Next, variables are extracted from the reference code for the question. These two sets are then combined to form a more robust reference set, as it captures the context of the question along with the tutor's expertise in naming variables.

A similar approach is taken with the student code to identify and normalize its variables, which makes it possible to compare them with those in the reference set.

Student variables that do not match the reference set are cataloged as inappropriate. These variables are also subjected to a more trivial analysis to check conformities and conventions of the python language, being classified as inadequate if they diverge from these standards.

Then, for each inappropriate variable, a syntactic and semantic similarity analysis carried out in relation to the variables in the reference set, generating a list of variables with their respective similarity scores.

The selection of substitute variables is defined based on the configuration of a pre-defined threshold; candidate variables that meet or exceed this threshold are chosen as appropriate surrogates. If no candidate variable meets the threshold criteria for a given unsuitable variable, all reference variables are considered as replacement options. As result, each unsuitable variable will have a set of substitute variables.

3.3 Feedback

Once the inappropriate variables in the student's code and the candidates for replacements have been identified, suggestions regarding the code are then generated, which are then forwarded to Cosmo to be presented to the students. The student may or may not consider them when refactoring their solution. Figure 1 and 2 shows some messages and how they are arranged for the student.

The messages can be presented in two ways, which refer to the location where the messages will be displayed. These forms are related to the scope of respective code smell. Code smells that refer to a specific construction or declaration on a line are marked with icons indicating their severity and importance. These include code smells related to variable names, code structure, language best practices, and are mapped directly onto the code itself. The other form has broader scope and coverage, relating to code complexity, whether it's cyclomatic complexity, number of lines, or Halstead metrics, compared to reference code, and they are displayed just below, outside the coding area.

The code smells mentioned related to the line scope receive symbols and colors referring to their severity and importance and can be of three types:

Red – Indicates problems related to language standards/conventions, especially in relation to variable names (snake_case, camelCase, upper and lower case). These problems are strongly recommended for refactoring.

Yellow – Represents warnings related to community standards regarding the use of code constructs and language design It is also associated with suggesting more descriptive and representative variable names. These are problems that require attention, and refactoring is advisable.

Blue – Provides information related to additional tips and guidance. It works mainly to reinforce the use of general and recurring good coding practices.

4 EVALUATION

To investigate student behavior in relation to feedback generated by CosmoMentor, we adopted a predominantly quantitative approach, centered on the analysis of objective data. To do this, we collected 467 log data from 36 students while the tool for 4 weeks and performed statistical analyzes and graphs to verify the relationships between the variables involved.

4.1 Participants

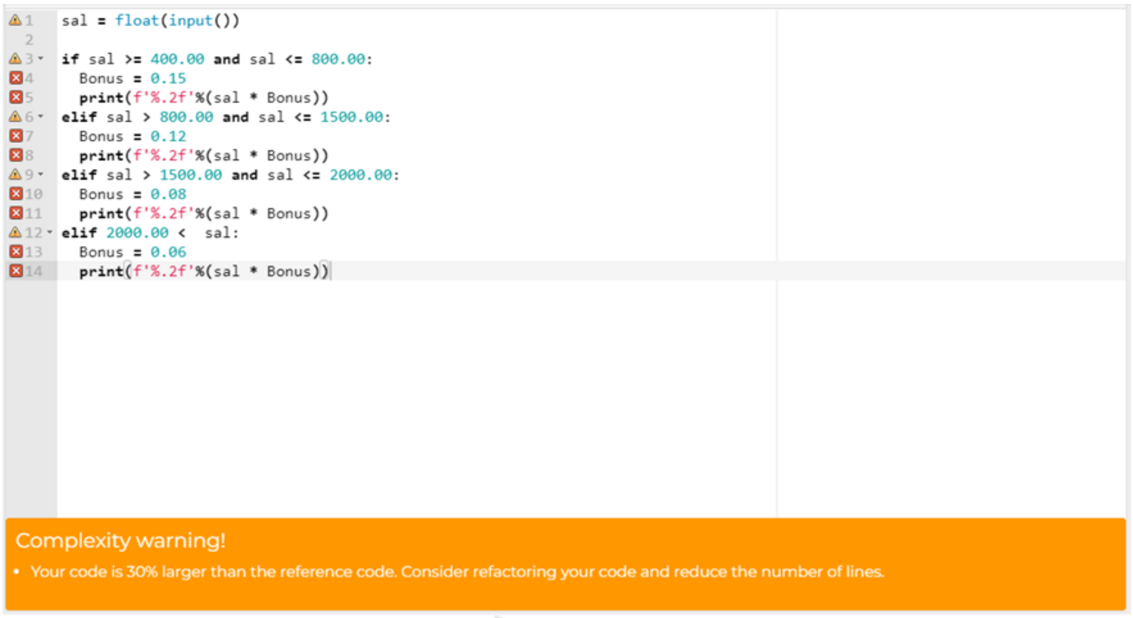
The research involved 36 students enrolled in the Algorithms I discipline in a Computer Science course during the second semester of 2023.

Regarding the academic profile, the sample includes first-year students. Most participants had little to no experience in programming and algorithmic logic. This intentional characteristic of the sample sought to capture the usage pattern of the tool by beginner students.

Data collection took place through a voluntary approach, where students were invited to participate in the research during regular classes in an Algorithm discipline. This recruitment method sought to guarantee a representative sample of the specific academic context, minimizing possible selection biases.

4.2 Dataset

The data was collected by recording students' interactions with the feedback tool as they submitted their solutions to questions in the Cosmo environment

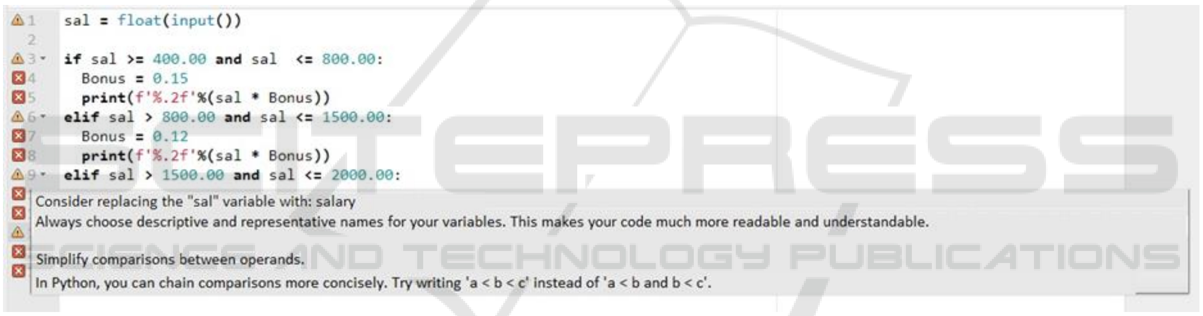


```
1   sal = float(input())
2
3   if sal >= 400.00 and sal <= 800.00:
4       Bonus = 0.15
5       print(f'%.2f'%(sal * Bonus))
6   elif sal > 800.00 and sal <= 1500.00:
7       Bonus = 0.12
8       print(f'%.2f'%(sal * Bonus))
9   elif sal > 1500.00 and sal <= 2000.00:
10      Bonus = 0.08
11      print(f'%.2f'%(sal * Bonus))
12  elif 2000.00 <  sal:
13      Bonus = 0.06
14      print(f'%.2f'%(sal * Bonus))
```

Complexity warning!

- Your code is 30% larger than the reference code. Consider refactoring your code and reduce the number of lines.

Figure 1: Cosmo code editor integrated with messages generated by CosmoMentor for salary bonus calculation code.



```
1   sal = float(input())
2
3   if sal >= 400.00 and sal  <= 800.00:
4       Bonus = 0.15
5       print(f'%.2f'%(sal * Bonus))
6   elif sal > 800.00 and sal <= 1500.00:
7       Bonus = 0.12
8       print(f'%.2f'%(sal * Bonus))
9   elif sal > 1500.00 and sal <= 2000.00:
```

Consider replacing the "sal" variable with: salary
Always choose descriptive and representative names for your variables. This makes your code much more readable and understandable.

Simplify comparisons between operands.
In Python, you can chain comparisons more concisely. Try writing 'a < b < c' instead of 'a < b and b < c'.

Figure 2: Messages for line nine regarding the description and representation of the “sal” variable, in addition to suggestions related to the construction of the comparison structure between operations.

over 4 weeks. Each interaction was captured, including the set of feedback messages provided for each feedback class and the students’ solutions.

The analysis was carried out on this subset because the research intention was to evaluate whether students take into consideration the qualitative feedback suggestions provided by CosmoMentor in solving the activities, even after achieving functionally correct solutions, improving their code beyond the functional aspect.

4.3 Data Analysis Methods

In order to achieve the research objectives, we conduct statistical analyzes that include both correlation tests between variables and their visualization for a more in-depth understanding of the results.

Before conducting the mentioned procedures, we initially performed a visualization of the number of student resubmissions, and subsequently identified the associated outliers and removed them from the correlation analysis. Figures 3 and 4, respectively, show these visualizations.
To investigate the relationship between variables in our study, we chose to use Spearman correlation instead of other correlation measures such as Pearson correlation. This decision was based on specific considerations related to the nature of the data. Spearman’s correlation is a non-parametric measure of association between two variables that does not require the assumption of normal data distribution, which is exactly what we have in figure 3.

Therefore, we performed Spearman correlation tests between the key variables that define the classes of code smells and the number of student resubmissions. Additionally, we assessed the statistical significance of this correlation using the associated hypothesis test, with a significance level set at α = 0.05.

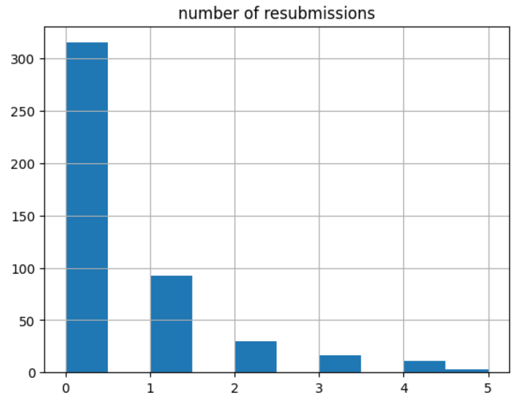


Figure 3: Number of resubmissions.

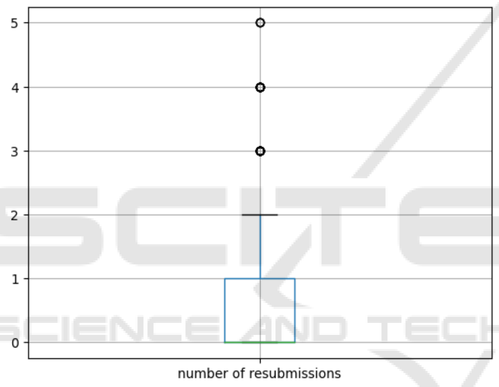


Figure 4: Outliers for the number of resubmissions.

5 RESULTS

The results of the correlation analysis between the types of code smells (complexity, variable names, refactoring) and the number of student resubmissions revealed significant insights into the relationship between code characteristics and the resubmission process.

5.1 Participants' Behavior

Our research aimed to investigate whether students take into account the qualitative feedback suggestions given by CosmoMentor when solving activities. Therefore, correlation analysis allowed us to explore these relationships and contribute to the understanding of the factors that impact the use of the tool and the quality of the code. Table 1 shows the correlation coefficients between these variables.

The correlation analysis revealed a significant correlation between the code smells variables related to variable name and number of resubmissions ($r = -0.4718$, $p < 0.05$), indicating a negative relationship between these variables.

This suggests that the quality of variables naming tends to improve as the student receives qualitative feedback and refactors their code. Figure 5 helps us see this behavior.

It is possible to observe that as students resubmit their codes, problems related to variable naming tend to decrease. This shows that, at least for codes smells related to variable names, CosmoMentor is capable of influencing and improving the quality of beginner students' code.

However, some correlation tests did not show significant relationships. For example, no significant relationship were found between code smells related to complexity and the number of resubmissions, nor were they found for code smells related to refactoring and the number of resubmissions.

Table 1: Correlations between the types of code smells and the variables analyzed.

Variable	Correlation coefficient (r)
Complexity	-0.0290
Variable Name	-0.4718
Refactoring	-0.0682

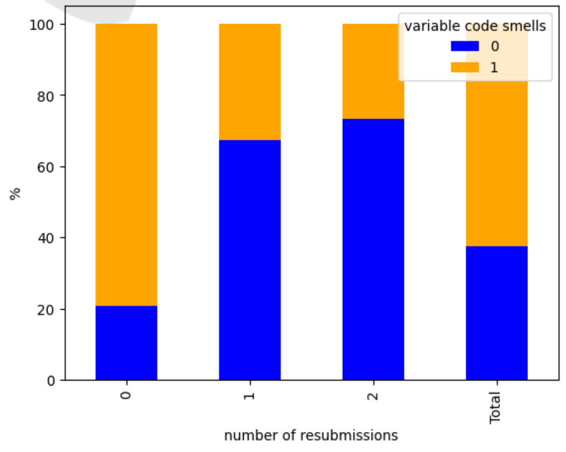


Figure 5: Percentage of code smells for variables by resubmission number.

These non-significant correlations suggest that others unexplored variables may be playing an important role in these relationships. Possible limitations, such as sample size or lack of control for confounding variables, may influence the strength of the observed associations.

One hypothesis for this lack of correlation is the possibility that problems related to complexity and refactoring are too complex for beginner students to solve on their own, or even that the suggestions presented for these code smells are not enough for students to be able to refactor their codes.

6 CONCLUSIONS

In this study, we developed a qualitative feedback system and investigated its ability to influence and improve the quality of beginning students' code, analyzing the correlation between resubmissions to the system and some classes of code smells. Our results provide positive insights into the use of the system to improve the quality of student codes, showing great promise in the task of refactoring variable names.

During the study, we observed a significant correlation between the number of resubmissions to the system and the number of code smells related to the nomenclature of variable names in the students' code.

Despite the promising results obtained in this study, it is important to recognize some limitations that may influence our conclusions. Firstly, the sample used in this study was restricted to a single class of algorithms, which may limit the generalization of the results to other populations. Furthermore, due to the nature of longitudinal design adopted, we can only capture participant behavior during this specific period, without the ability to observe long-term changes or usage patterns. This may limit our understanding of tool usage trends over time and its long-term sustainability. Finally, it is important to note that external factors, such as changes in participants' individual circumstances or unforeseen events, may have influenced the use of the tool throughout the study period.

Ultimately, our study highlights the importance of automated qualitative feedback related to code quality in online judge environments as a practical intervention to promote evaluation of novice students' codes beyond the functional. We hope that this study inspires other researchers to contribute even more to this still little explored field.

ACKNOWLEDGMENTS

We would like to acknowledge CAPES for the financial support.

REFERENCES

Araujo, E., Serey, D., & Figueiredo, J. (2016, October). Qualitative aspects of students' programs: Can we make them measurable?. In *2016 IEEE Frontiers in Education Conference (FIE)* (pp. 1-8). IEEE.

Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., ... & Bryksin, T. (2022, February). Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1* (pp. 307-313).

Chren, S., Macák, M., Rossi, B., & Buhnova, B. (2022, June). Evaluating code improvements in software quality course projects. In Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering (pp. 160-169).

Fowler, M. (1999). Refactoring: improving the design of existing code. Addison-Wesley Professional.

Jiang, L., Rewcastle, R., Denny, P., & Tempero, E. (2020, June). Comparecfg: Providing visual feedback on code quality using control flow graphs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 493-499).

Keuning, H., Heeren, B., & Jeuring, J. (2020, November). Student refactoring behaviour in a programming tutor. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (pp. 1-10).

Liu, X., & Woo, G. (2020, February). Applying code quality detection in online programming judge. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology* (pp. 56-60).

Orr, J. W. (2020, novembro). Automatic assessment of the design quality of student python and java programs. J. Comput. Sci. Coll., 38(1), 27-36.

Ureel II, L. C., & Wallace, C. (2019, February). Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 738-744).

Wasik, S., Antczak, M., Badura, J., Laskowski, A., & Sternal, T. (2018). A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, *51*(1), 1-34.

Zhou, W., Pan, Y., Zhou, Y., & Sun, G. (2018, May). The framework of a new online judge system for programming education. In *Proceedings of ACM turing celebration conference-China* (pp. 9-14).