Investigating Code Smells in K-12 Students' Programming Projects: Impact on Comprehensibility and Modifiability

Verena Gutmann[0009−0007−2892−9160], Elena Starke[0009−0004−5100−368X], and Tilman Michaeli[0000−0002−5453−8581]

Technical University of Munich, Arcisstraße 21, 80333 Munich, Germany
{verena.gutmann,elena.starke,tilman.michaeli}@tum.com

Abstract. Teaching students to code goes beyond focusing on the correct implementation of features. It is also about emphasizing the importance of comprehensible and modifiable code. Addressing these characteristics is already crucial for novice programmers in K-12 computing education, as fostering code quality can support their learning process especially when working collaboratively in group projects. This study aims to investigate the extent to which code smells are problematic in K-12 students' code and understand the impact of different code smells on comprehensibility and modifiability. We initially selected relevant code smells to address this research objective and then conducted a qualitative analysis of 12 student projects. The results show a differentiated picture for different types of code smells. While *Duplicated Code* and *Class Data Should Be Private* may not be critical issues. However, in our data, *Long Functions*, *Speculative Generality*, *Comments*, *Mysterious Names*, and bad *Code Formatting* negatively affected the comprehensibility and modifiability.

Keywords: novices · K-12 · computing education · code smells

1 Introduction

Learning to program is one major challenge in the K-12 computing education classroom. While teaching novices the fundamentals of programming syntax and implementing desired functionality is essential, it is equally crucial to recognize the significance of fostering code quality to enhance the learning process, particularly in group work projects.

Looking into professional software development, besides correct functionality, it is equally important to consider other quality aspects, such as maintainability, testability, comprehensibility, or modifiability. These factors are crucial in facilitating collaboration among developers and ensuring code's long-term viability and maintainability throughout the development process [15].

One key factor that can harm code quality is the presence of code smells [4]. Code smells are not related to functionality or syntax but serve as indicators of potential issues within the code. They act as warning signs, highlighting areas

of code that may require improvement. Comprehensibility and modifiability are particularly susceptible to the harmful effects of code smells and are consequently relevant to consider in collaborative work.

Preparing students to become future professional developers is no goal of K-12 programming education. Nevertheless, it remains crucial to emphasize the significance of writing comprehensible and modifiable code, particularly in collaborative settings. Code smells, which impact comprehensibility and modifiability, can challenge effective collaboration among students. Therefore, addressing code quality and mitigating the presence of code smells can enhance the collaborative learning experience in K-12 programming education. Furthermore, however, it is crucial to enable students to understand how and why a program works or does not work and to be able to comprehend their code. Additionally, students often need help to solve issues independently and sometimes even make it more complicated than it has to be [7].

While existing research has extensively explored code smells in various programming contexts, the specific domain of K-12 students' programming projects still needs to be explored. Therefore, in this paper, we investigate the impact of code smells on comprehensibility and modifiability to address them adequately in the classroom setting.

2 Theoretical background and related work

It is challenging to define comprehensibility in the context of source code due to its subjective nature. Code comprehensibility is closely related to code readability – "a human judgment of how easy a text is to understand" – which can be considered a prerequisite for comprehension [3]. The aim of comprehensibility should be to make it as effortless as possible for readers and future editors to familiarize themselves with the code. A clear structure that includes a meaningful arrangement of instructions and method calls, and uniform code formatting can be helpful [8]. Additionally, self-explaining variable names, simple control structures, and good documentation can support code maintenance, resulting in improved modifiability [3].

In contrast, code smells are recognized patterns in source code that indicate potential areas for improvement rather than bugs or errors. They act as indicators within object-oriented code, drawing attention to areas of weakness that could benefit from closer examination and refactoring. By identifying code smells, developers can proactively address issues and enhance the overall quality of their code. Various taxonomies have been proposed to classify these smells, including inappropriate comments, excessively long functions, or an over-reliance on primitive data types [4]. In addition to code smells, Stegeman et al. propose a rubric for educational settings to assess code quality and promote effective coding practices in novice programming courses. Considering aspects relevant for beginners, this rubric encompasses ten categories grouped into four overarching categories: *Documentation*, *Presentation*, *Algorithms*, and *Structure* [19, 20].

Previous research on quality issues in K-12 education has predominantly focused on block-based programming and tools such as Scratch [6]. In contrast, text-based programming has mainly been studied in university settings.

Looking into block-based programming, there are various findings regarding common issues. The source code of Scratch programs created by high school students is frequently characterized by multiple code smells, such as problematic variable names and duplicated code, which can negatively affect both correctness and readability [5]. Furthermore, students often acquire certain programming habits that can result in code smells. The prevalence of extensive bottom-up programming and extremely fine-grained programming approaches can result in code smells like dead code and an overwhelming number of scripts. Consequently, students may face significant challenges when debugging and maintaining their projects, as the complexity and scale make these tasks practically impossible [14]. Even when the students' projects were not particularly complex, those code smells arise [1]. Additionally, code smells such as too-long methods and code duplications impacted students' performance negatively. The *long method* smell hindered their understanding, while duplication decreased the modifiability of their code [9]. Even if comprehensibility and modifiability seem rather abstract, K-12 students can understand aspects of software quality just as well as general programming concepts [10]. In addition to examining poor programming habits, there is a potential approach to address code quality in K-12 classrooms by emphasizing "code perfumes" that represent good programming practices. Specific structures like *parallelism*, *nested loops*, and *nested conditional checks* are identified as beneficial patterns that are often found in functionally correct code [17].

Considering text-based programming in university settings, evaluating student projects led to identifying several quality issues. Using Java, the most common code smells were missing blank lines and a quirky usage or omission of parentheses. Considering Python programs, spaces were often omitted after the comment character or lines contained too many characters [11]. Overall, the correction rate for quality issues among students is low, with many topics going unnoticed or unaddressed even when using tools designed to detect them [12]. The comparison of static quality properties between first and second-year college students revealed that second-year students exhibited improvements in certain aspects, such as using shorter functions and fewer very short variable names, compared to their first-year counterparts. However, they also tended to write more complex code and incorporate a higher level of statement nesting within methods. Despite these differences, there was no significant improvement in the overall code quality of second-year students compared to first-year students. This indicates that code quality skills only improve by explicitly addressing them [2].

In summary, considerable research has been conducted on code quality in novice programming education, primarily focusing on university-level settings. Considering K-12, existing research in this domain predominantly centers around block-based programming languages such as Scratch. As a result, there is a need

for further investigation and exploration of code quality aspects, specifically in text-based programming within K-12 educational contexts.

3 Methodology

To address the research gap described above, we aim to investigate the source code of upper secondary students in this study. To gain a comprehensive understanding of the issues posed by code smells in schools, our approach diverges from a mere examination of their frequency. Rather, we strive to delve deeper, investigating the extent of the impact caused by these code smells. To this end, we address the following research question: *How do code smells impact the comprehensibility and modifiability of source codes of programming novices in upper secondary school?*

3.1 Sample

To answer our research question, we analyzed a total of $N = 12$ source codes created by students as part of group work projects towards the end of the school year. The projects were collected from different German High Schools: four were created by year 11 students (P1 to P4) and eight by year 10 students (P5 to P12). At this point, students have one or two years respectively of programming experience in a text-based language. All projects implement small games programmed in Java using BlueJ, a widely utilized development environment designed for novices. The source codes from the 11th-grade class represent intermediate versions of their respective projects. However, it is noteworthy that all projects can be executed without errors. The length of the source code ranges from 161 to 735 non-empty lines. The mean is 325.5 lines of code, with a median of 292 lines.

3.2 Data Analysis

We conducted a structured qualitative content analysis according to [13] to analyze the data. We used a deductive category system based on various catalogs for code smells. However, due to the relatively small and less intricate nature of projects at this level, it is worth noting that several of the defined code smells aimed at professional development are not applicable in the school context. Based on these considerations as well as findings in related work, we have chosen the following set of code smells from [4]: *Long Function*, *Duplicated Code*, *Comments*, *Mysterious Name* and *Speculative Generality*. Furthermore, we included the code smells *Class Data Should Be Private* [18] and *Code Formatting* [20] in our analysis. Table 1 lists the complete category system.

To detect these code smells, we have established specific thresholds that we consider meaningful for school projects with limited scope and complexity. A function, therefore, is a *Long Function* if it exceeds 20 lines of code. In the case

of *Duplicated Code*, at least three lines must be duplicated, even if the variable names differ. Regarding *Comments*, they are flagged as code smells if they merely describe the code without providing any additional information, in line with the definition by [4]. However, commented-out lines of code are not considered code smells. The evaluation of the *Mysterious Name* code smell is based on the Oracle Naming Conventions[1]. However, the distinction between uppercase and lowercase letters is not taken into account, as it does not significantly impact the code's comprehensibility or modifiability. Some analyzed projects are incomplete, so the code smell *Speculative Generality* refers only to unused attributes, variables, and parameters. Guidance on capturing the code smell *Code Formatting* is provided by the criteria for Java from Checkstyle[2].

Table 1. Final Category System

name	**description**
Long Function (LF)	A method has huge size.
Duplicated Code (DC)	A code section is included multiple times.
Comments (C)	A comment is superfluous.
Mysterious Name (MN)	The name of a variable, class, or method is not self-explanatory.
Speculative Generality (SG)	A part of the code is not called.
Class Data Should Be Private (CDSP)	A class exposes its attributes.
Code Formatting (CF)	The formatting of the code is not clear.

The detection of code smells was automatically done using the IDE IntelliJ[3]. This involved using the integrated code inspection and analysis, as well as the Checkstyle[4] and Statistic[5] plugins. If necessary, manual evaluation was performed when automated detection was not feasible. For all individual code smells, we evaluated their actual impact on the modifiability and comprehensibility of the source code individually based on a qualitative interpretation in the context of the related project. Given the difficulty and subjective nature of measuring these aspects, in the following, we describe in detail our interpretation and reasoning resulting from discussions in our research group.

4 Results

To first of all provide an overview of the smells in all projects, see Table 2. We employed two metrics to measure the number of detected code smells. For

[1] https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
[2] https://checkstyle.sourceforge.io/checks.html
[3] https://www.jetbrains.com/de-de/idea/
[4] https://plugins.jetbrains.com/plugin/1065-checkstyle-idea
[5] https://plugins.jetbrains.com/plugin/4509-statistic

code smells that tend to increase with code length, we express the frequency as occurrences per 100 lines (1). For code smells specific to a particular program element, we present the proportion of code smell occurrences relative to the total number of program elements of the corresponding type (2) [21].

Table 2. Frequencies of code smells by projects studied. "–" marks that no comments exist in the code, while 0% means that none of the dedicated program elements is considered a code smell.

	metric	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
LF	(1)	18	9	27	23	41	17	11	41	0	24	35	40
DC	(1)	0	3	3	2	5	22	3	0	0	8	23	0
C	(2)	72%	82%	30%	90%	-	100%	-	100%	100%	-	100%	100%
MN	(2)	8%	30%	23%	13%	29%	41%	3%	50%	28%	8%	29%	36%
SG	(1)	8	10	5	8	22	5	7	3	8	10	5	0
CDSP	(2)	40%	75%	40%	33%	0%	58%	10%	72%	100%	0%	0%	0%
CF	(1)	60	54	72	76	40	189	21	11	25	42	47	40

Long Function 22 methods were identified as *Long Function*. Half are slightly above the threshold and, therefore, less critical concerning comprehensibility and modifiability than the others. However, one method in particular (P8) spans 120 lines, accounting for 34% of the total code. This code lacks structure and clarity, making it prone to complications during modification. Notably, many very long functions are responsible for updating the user interface. Particularly conspicuous is such an expansion of functions in the projects of year ten students. Additionally, constructors (P2, P4, P10, P11) are particularly susceptible to becoming lengthy. Another factor contributing to long functions is the presence of if-statements (P11) which have empty branches. Furthermore, extensive if-statements with multiple cases can also contribute to the length of functions, but they remain readable due to their structured nature (P7).

Finding: Functions only slightly above the threshold do not pose significant limitations on comprehensibility and modifiability. However, in our data, some other functions lack a clear structure due to their excessive length and consequently impact comprehensibility and modifiability.

Duplicated Code Overall, a relatively low number of *Duplicated Code* instances was detected. The longest duplicate consists of only 11 lines (P11), and many duplicates are slightly above the threshold (P3, P4, P7, P10), indicating no significant issue regarding comprehensibility. The duplicates could be easily refactored in several cases by extracting them into separate functions (P3, P4, P5, P6, P11) to enhance the code's comprehensibility and structure. Additionally, adding parameters could combine some methods (P2, P6). In one outstanding example, three functions perform the same task but have different names (P6). Modifying one function requires changing the other two as well. Another critical

instance of duplication arises from identical function calls within an if-statement under different conditions (P10), with the only difference being the passed argument (in this case, the color of an object as a String). Changing the method name would force multiple changes in the condition branches. One *Duplicated Code* smell requires advanced knowledge for refactoring. It involves code sections in four different classes (P6), where the constructors share identical code and contain another method with the same code. As already mentioned above, one change would entail several modifications.

Finding: *Duplicated Code* is relatively scarce overall, and when it is, it only slightly complicates comprehensibility. Concerning modifiability, only some critical code smells could mostly be addressed by simple refactoring.

Comments In projects P5 to P11, only a few comments are used. The existing comments are obviously from code templates that have not been customized and, therefore, cannot be considered beneficial and thus count as a code smell. However, it is worth appreciating the inclusion of author information and the providing short and concise descriptions of classes and functions, which helps in collaboration (P3). Comments that solely describe the functionality of a method, such as "closes the database connection" preceding a function named *ConnectionClose* (P1), do not offer significant additional support for comprehension and are consequently considered as a code smell. In some projects, every line of code was provided with a comment explaining the corresponding line (P2). This leads to a rather unattractive and difficult-to-survey picture of the code. Another issue that could be observed is that *Comment* code smells were also apparently created in the comments by copying them, causing incorrect and useless information (P2). Furthermore, in an if-statement, comments repeating the conditions were identified (P4). This information is redundant for the reader and makes it difficult to read.

Finding: Comment usage among students varies greatly. While some students refrain from using comments, others use them excessively without adding much value for comprehensibility.

Mysterious Name This code smell is present in every project. Generally, a mixture of German and English is commonly used, which only somewhat impacts comprehensibility. However, poor naming can negatively affect comprehensibility, even though its impact on modifiability is not crucial. Overall, the methods are mainly named understandably. It should be noted that we have not extensively investigated the functionality of the methods, and therefore may be discrepancies between the naming and the actual functionality. However, method names *niceMethod* (P10) and *addtt* (P2) do not provide any indication of their functionality. Regarding class names, the only related smells found in the data are the designations *LE* and *MyKeyAdapter* (P2). The variable naming reveals a contrasting scenario as it often lacks meaningful names. Despite being short, these names only sometimes convey the variable's functionality effectively to observers. Frequent usage of single-letter designations can lead to confusion, mainly

when numbering is utilized for differentiation (P1, P2, P3). Numbering is predominantly done for naming user interface objects (P5, P8, P11, P12) obscuring the association between specific objects and their corresponding graphical task. Additionally, certain abbreviations, which should ideally be avoided according to naming conventions, are frequently employed in numerous projects. Some of them, such as *koord* (P3), *min* (P6), or *xpos* (P2), can be derived by a reader and can be considered less critical. In contrast, there are also less comprehensible namings such as *sadpicm, sadpicq* (P6), *tt* (P2), or *LHor* (P9).

Finding: Most functions and classes have expressive names, enhancing code comprehensibility. However, variables often need more precise naming, relying on abbreviations or numbering that hinder code comprehensibility and impair collaboration. Nevertheless, this does not directly impact modifiability.

Speculative Generality In general, it is worth mentioning that the projects examined had relatively few unused components. As a result, these unused components only have a minor impact on the code comprehensibility and modifiability. Among these, unused parameters and attributes are the least critical, as they do not disrupt the overall flow of the program. However, they can introduce confusion when unused attributes result from the presence of local variables of the same type within class methods (P2, P6). Likewise, initializing unused local variables (P3, P6, P9, P10) can hinder comprehensibility. In one project (P10), an integer variable is declared before a for-loop, seemingly meant to serve as a constraint for the loop's iteration. However, the hard-coded value is used within the loop instead of the variable. This particular example may not be of utmost importance, but it has the potential to cause some irritation.

Finding: The code smell *Speculative Generality* is generally a minor issue that has a limited impact on code comprehensibility and modifiability. Solely, unused local variables are more likely to affect code comprehensibility negatively.

Class Data Should Be Private Four of the examined source codes do not exhibit inappropriate data encapsulation. However, in other projects (P1, P6, P8, P9), more than half of the attributes raised concerns as code smells. Some classes contain either only *public* attributes (P8) or do not include any attributes marked with the *private* access modifier (P9). In most cases, the attributes lacked an explicit access modifier, making them accessible to other classes within the same package. It is important to note that the investigated projects did not utilize packages, resulting in all classes having access to these attributes. The code smell *Class Data Should Be Private* does not directly contribute to poor code comprehensibility or pose challenges for modifiability.

Finding: The improper usage of access modifiers is typically a minor and isolated issue that does not significantly impact the comprehensibility or modifiability of the code itself.

Code Formatting Significant instances of poor *Code Formatting* were detected in all the analyzed source codes. Remarkably, in one particular project, there were approximately 189 violations of Checkstyle rules per 100 lines of code. The

most common violations in all projects involve missing spaces around operators, although these have a relatively minor impact on the overall code comprehensibility. Violations such as missing spaces after a comma or between the method name and parameter list are also relatively unproblematic. However, instances of curly braces placed on the wrong line are more significant for maintaining a clear code structure, frequently occurring in the analyzed source codes. Although consistent misplacement may not severely impact comprehensibility, it can lead to challenges when making modifications. Similarly, the absence of optional curly braces, although less frequent, is critical for code modification. Omissions and misplacement of braces can result in errors when inserting code in the wrong location, consuming substantial time to identify.

Finding: Cluttered and inconsistent code formatting reduces comprehensibility and poses a risk for future modifications.

5 Discussion

In this study, we qualitatively analyzed students' source code to investigate how selected code smells influence the comprehensibility and modifiability of code written by novice programmers. To this end, we used a selection of typical code smells from professional software development. Although we have great differences in the scope and aims of programming projects in K-12, using those professional patterns is a typical approach for analyzing students' projects. In line with this, our results indicate that certain smells, such as *Mysterious Name* and *Code Formatting*, are severe issues in the students' code. However, others, such as *Class Data Should Be Private*, do not seriously affect the comprehensibility and modifiability of the high-school students' projects.

Consequently, we consider addressing the problem of *Mysterious Names* in the classroom as highly relevant, mainly when students work collaboratively in teams. Notably, the prevalence of poorly named variables is not limited to specific programming languages, as similar problems can be observed in Scratch projects [16].

Comments can help in keeping an overview, especially when working on a project over a longer period of time. In addition, in team settings, good comments are crucial for collaboration. Interestingly, our analysis revealed that several comments in the code lack meaningful or new information, appearing to be included solely so that the code contains comments, possibly to comply with specifications. Following the agile principle using code as documentation, excessive comments could be reduced, thus well-chosen variable and method names, thereby addressing the *Mysterious Name* code smell.

We were surprised to find a high occurrence of cluttered *Code Formatting* in all the projects because a Checkstyle-plugin for BlueJ is available. BlueJ also provides automatic formatting features for consistent and clean code. Although missing blank lines and parentheses were frequently observed in other studies [11], we found missing parentheses were less common in our results. Instead,

missing spaces and the position of curly braces were more prominent issues affecting code quality.

Concerning the code smell *Speculative Generality*, we hypothesize that in many cases, it can be attributed to oversight or carelessness, where elements were not removed when they became unnecessary. Interestingly, contrary to findings in Scratch projects [1], the results of this study suggest that unused code may be considered less problematic.

Our findings regarding the code smell *Duplicated Code* also differ from previous results, specifically with studies focusing on frequently occurring code duplication in Scratch projects [1, 16]. Several factors, including the age of the students, the distinction between text-based and block-based languages, and the presence of an online repository for Scratch projects, could potentially account for these disparities. However, despite the ease of resolving the detected duplicates and their minimal impact on comprehensibility and modifiability within our data, addressing the issue of redundant code in school can still be valuable, particularly in collaborative group projects. Looking at the individual smells, it becomes apparent that the sense of certain smells, such as *Speculative Generality* and *Duplicated Code*, might be difficult for students to grasp, particularly in understanding their impact on code modifiability. This difficulty can be attributed to the relatively short and limited scope of school projects, which may not provide a broad context for students to grasp the significance of modifiability.

We want to emphasize that a high frequency of a code smell does not always indicate a high impact in our specific context. For example, the occurrence of *Mysterious Names* is less frequent than misplaced *Comments*, but it has a higher impact on comprehensibility. However, the effect of inappropriate comments may vary depending on the specific case.

5.1 Limitations

We used a qualitative approach to gain deep insights into selected code smells and assessed their impact individually. Consequently, our data and results are not necessarily representative for the target group. Additionally, it is worth noting that assessing the impact on comprehensibility and modifiability is inherently subjective. Furthermore, we lack direct insight into the programming classes and have no access to information regarding the guidance teachers provide. Therefore, we have limited information regarding conventions and evaluation principles presented to the students, making it challenging to draw conclusions about the reasons for occurring code smells.

6 Conclusion

In this study, we investigated code quality in novice programmers' projects, specifically focusing on how code smells impact code comprehensibility and modifiability. To this end, we conducted a qualitative content analysis of group projects of novice programmers. Our results show that several code smells occur

in students' source code. For example, *Mysterious Name* and *Code Formatting* are regarded as severe issues in the students' code, especially when working together on group projects. In our data, other code smells, like *Class Data Should Be Private* and *Duplicated Code*, do not seriously affect comprehensibility and modifiability.

The findings of this study contribute to our understanding of the impact of code smells on collaborative programming projects in K-12 computing education classes. To broaden our subjective view in further research, it could be interesting to evaluate the students' perspective on the code quality of their projects and investigate whether and how code quality depends on the group size. Furthermore, our results raise questions about how to effectively address them in the classroom and communicate their importance to K-12 students. One approach might be to incorporate activities that promote code quality, such as refactoring, using a linter, or employing an appropriate code-evaluation rubric. By integrating such activities into teaching, e.g., a specific phase for refactoring within project-based learning, students can gain hands-on experience in improving code quality that supports them, particularly in collaborative settings. Additionally, these activities might provide a valuable opportunity for novice programmers to enhance their programming competencies as they delve into the code and actively work towards improving its quality.

References

1. Aivaloglou, E., Hermans, F.: How kids code and how we know: An exploratory study on the scratch repository. In: Proceedings of the 2016 ACM conference on international computing education research. pp. 53–61 (2016)
2. Breuker, D.M., Derriks, J., Brunekreef, J.: Measuring static quality of student code. In: Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. pp. 13–17 (2011)
3. Buse, R.P., Weimer, W.R.: Learning a metric for code readability. IEEE Transactions on software engineering **36**(4), 546–558 (2009)
4. Fowler, M.: Refactoring, Improving the Design of Existing Code. Addison-Wesley, 2 edn. (2019)
5. Frädrich, C., Obermüller, F., Körber, N., Heuer, U., Fraser, G.: Common Bugs in Scratch Programs. In: Giannakos, M., Sindre, G., Luxton-Reilly, A., Divitini, M. (eds.) Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. pp. 89–95. ACM, New York, NY, USA (2020)
6. Fraser, G., Heuer, U., Körber, N., Obermüller, F., Wasmeier, E.: Litterbox: A linter for scratch programs. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET). pp. 183–188. IEEE (2021)
7. Gugerty, L., Olson, G.: Debugging by skilled and novice programmers. SIGCHI Bulletin **17**(4), 171–174 (apr 1986)
8. Hansen, M., Goldstone, R.L., Lumsdaine, A.: What makes code hard to understand? arXiv preprint arXiv:1304.5257 (2013)
9. Hermans, F., Aivaloglou, E.: Do code smells hamper novice programming? a controlled experiment on scratch programs. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). pp. 1–10. IEEE (2016)

10. Hermans, F., Aivaloglou, E.: Teaching software engineering principles to k-12 students: a mooc on scratch. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET). pp. 13–22. IEEE (2017)
11. Karnalim, O., Chivers, W., et al.: Work-in-progress: Code quality issues of computing undergraduates. In: 2022 IEEE Global Engineering Education Conference (EDUCON). pp. 1734–1736. IEEE (2022)
12. Keuning, H., Heeren, B., Jeuring, J.: Code quality issues in student programs. In: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. pp. 110–115 (2017)
13. Mayring, P.: Qualitative content analysis: A step-by-step guide. Sage (2022)
14. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.: Habits of programming in scratch. In: Proceedings of the 16th ACM conference on Innovation and technology in computer science education. p. 168–172. Association for Computing Machinery, New York, NY, USA (2011)
15. Mistrik, I., Soley, R.M., Ali, N., Grundy, J., Tekinerdogan, B.: Software quality assurance in large scale and complex software-intensive systems. Morgan Kaufmann (2016)
16. Moreno, J., Robles, G.: Automatic detection of bad programming habits in scratch: A preliminary study. In: 2014 IEEE Frontiers in Education Conference (FIE) Proceedings. pp. 1–4. IEEE (2014)
17. Obermüller, F., Bloch, L., Greifenstein, L., Heuer, U., Fraser, G.: Code Perfumes: Reporting Good Code to Encourage Learners. In: Berges, M., Mühling, A., Armoni, M. (eds.) The 16th Workshop in Primary and Secondary Computing Education. pp. 1–10. ACM, New York, NY, USA (2021)
18. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 101–110. IEEE (2014)
19. Stegeman, M., Barendsen, E., Smetsers, S.: Towards an empirically validated model for assessment of code quality. In: Simon, Kinnunen, P. (eds.) Proceedings of the 14th Koli Calling International Conference on Computing Education Research. pp. 99–108. ACM, New York, NY, USA (2014)
20. Stegeman, M., Barendsen, E., Smetsers, S.: Designing a rubric for feedback on code quality in programming courses. In: Proceedings of the 16th Koli Calling International Conference on Computing Education Research. pp. 160–164 (2016)
21. Techapalokul, P., Tilevich, E.: Understanding recurring quality problems and their impact on code sharing in block-based software. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 43–51. IEEE (2017)