How Teachers Would Help Students to Improve Their Code

Hieke Keuning
Open University of the Netherlands and Windesheim University of Applied Sciences
hw.keuning@windesheim.nl

Bastiaan Heeren
Open University of the Netherlands
bastiaan.heeren@ou.nl

Johan Jeuring
Utrecht University and Open University of the Netherlands
j.t.jeuring@uu.nl

ABSTRACT

Code quality has been receiving less attention than program correctness in both the practice of and research into programming education. Writing poor quality code might be a sign of carelessness, or not fully understanding programming concepts and language constructs. Teachers play an important role in addressing quality issues, and encouraging students to write better code as early as possible.

In this paper we explore to what extent teachers address code quality in their teaching, which code quality issues they observe and how they would help novices to improve their code. We presented student code of low quality to 30 experienced teachers and asked them which hints they would give and how the student should improve the code step by step. We compare these hints to the output of professional code quality tools.

Although most teachers gave similar hints on reducing the algorithmic complexity and removing clutter, they gave varying subsets of hints on other topics. We found a large variety in how they would solve issues in code. We noticed that professional code quality tools do not point out the algorithmic complexity topics that teachers mention. Finally, we give some general guidelines on how to approach code improvement.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **Software engineering education**;

KEYWORDS

Programming education; code quality; refactoring

ACM Reference Format:
Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How Teachers Would Help Students to Improve Their Code. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319780

ITiCSE '19, July 15–17, 2019, Aberdeen, Scotland UK

ACM ISBN 978-1-4503-6301-3/19/07…$15.00
https://doi.org/10.1145/3304221.3319780

1 INTRODUCTION

An increasing number of studies have focused on the *quality* of programs written by novices, as opposed to the *correctness* of student programs, which has had quite some attention in research the past decades [7, 16]. These studies show that student programs contain a substantial amount of various quality issues, which often remain unsolved [8, 11, 15]. While there has not been much research into the reasons why quality issues remain unsolved, one can imagine that students are satisfied once their solutions pass all tests. They might not even be aware of quality aspects such as maintainability, performance and testability, or simply do not know how to satisfy them. Although a wealth of tools exists to analyse and refactor code, they are often not targeted at novices. Therefore, teachers play an important role in raising awareness of quality issues and encouraging students to improve functionally correct code.

There is little information on how to support students with improving the quality of their code, and what teachers consider to be a high-quality program. We conducted a study in which we collected this knowledge from experts. We asked 30 experienced educators who teach programming how they perceive the role of code quality in their courses. We showed them a number of functionally correct programs that have several issues related to quality, and asked them which hints they would give to help improve the program. We also asked them to describe the steps they would want the student to take to refactor the program into an improved version.

This paper (1) gives insight into how teachers assess the quality of novice programs, (2) shows how their hints compare to feedback generated by tools, (3) analyses how teachers would rewrite poor student code, and (4) describes how they would approach this rewriting in a stepwise way. These insights can be used to improve the development of courses and tools.

Section 2 gives some background and discusses related work. Section 3 describes the research questions, and how we collected and analysed the data. Section 4 shows the results for each research question, which are discussed in Section 5. Section 6 concludes and describes future work.

2 BACKGROUND AND RELATED WORK

This section establishes the meaning of central terms used in this paper and summarizes related work on code quality in education.

2.1 Code quality terms and definitions

Code quality deals with the directly observable properties of source code, such as algorithmic aspects (flow, expressions, language constructs) and structure (decomposition, modularization). Some examples of code quality issues are (1) duplicated code, (2) an expression that could be shortened, and (3) unnecessary conditional checks.

Although layout and commenting are certainly indicators of code quality, these aspects are beyond the scope of our study.

Fowler [10] uses the term *code smells* to describe characteristics in code that might indicate that something is wrong with the design of functionally correct code, which can have an impact on its quality. In the long term, low quality code may affect software quality attributes such as maintainability, performance and security. There are many tools available (e.g. PMD, SonarQube, Resharper, linters) to automatically detect quality issues and code smells in a program.

Code refactoring is improving code step by step while preserving its functionality. The well-known work by Fowler [10] describes a collection of refactorings, mainly focused on the structure of the code. Code Complete [13], a well-known handbook for software construction, describes refactorings on multiple levels: data-level (e.g. inline an expression), statement-level (e.g. use return instead of a loop control variable), routine-level (e.g. extract method), class implementation, class interface and system-level. Some IDEs offer support for refactorings, such as renaming variables and extracting methods. These IDEs execute a refactoring in a single step, which would not give novices much insight into how refactoring works.

An ITiCSE working group [5] investigated which quality aspects are considered important by teachers, students and developers. In our study we zoom in on how teachers assess the quality of student code, focussing on data-, statement- and routine-level refactorings, which are most relevant for the programs that beginners write.

2.2 Code quality in education

Multiple studies have investigated the quality of student programs. Pettit et al. [15] analysed submissions to an automated assessment system and found that several complexity metrics increased with every submission. Keuning et al. [11] detected many quality issues in over 2 million student programs, which were hardly ever fixed. Whether the student used a quality tool or not did not decrease the amount of issues. Breuker et al. [6] found no clear quality improvement between the code of first- and second-year students. De Ruvo et al. [8] investigated a set of 19.000 code submissions on 16 semantic style indicators, which address small issues such as unnecessary return statements, and too complex if-statements. They found instances in both code of novices and more experienced students. Luxton-Reilly et al. [12] investigated differences between correct solutions to programming exercises, identifying variation in structure, syntax and presentation. The authors found that even for simple exercises there are numerous variations in structure, and in some instances the teacher's solution was not the most popular one among students.

Although professional code quality analysers and refactoring tools are being used in education (e.g. [14]), there are also some tools designed specifically for education that give feedback on code quality, such as Style++ [2], FrenchPress [3], and AutoStyle [20]. AutoStyle gives stepwise feedback on how to improve the style of correct programs, based on historical student data.

Educators have designed several projects that teach students about refactoring, usually for more advanced students [1, 9, 17, 19]. Experienced educators studied the quality of object-oriented examples in Java textbooks [4]. They found several issues, in particular related to object-oriented thinking.

3 METHOD

The research questions this study addresses are:

RQ1 To what extent do teachers address code quality in their programming courses?

RQ2 What kind of hints related to code quality do teachers give to students, and how do these hints compare to the output of code quality tools?

RQ3 Which (stepwise) approach do teachers suggest to help students improve their programs, and what does the final improved program look like?

3.1 Study design

We designed a questionnaire[1] in which participants answer five questions about themselves and four short questions on the role of code quality. Next, we give our definition and scope of code quality and present three student programs of poor quality (see Section 3.1.1). We ask (1) how they would assess the program, (2) which hints they would give, and (3) how they would want the student to improve the program step by step, by typing the code after each step. We tested the questionnaire with a teacher who is not involved in this research, and adjusted the questionnaire according to his feedback.

We invited university teachers with at least two years of experience in teaching CS/programming-related courses to participate in our study. We did not ask professional programmers, because they do not necessarily have experience with teaching programming. We sent our invitation to 66 teachers from various institutes and countries, and asked them to forward the invitation to colleagues and other acquainted teachers.

3.1.1 Programs. The first exercise was taken from another study, including the most popular student solution (see Program 1) [12]. Its description is: 'Implement the sumValues method, which adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true.'

We designed the second exercise ourselves: 'Write the code for the method unevenSum. This method should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.' We collected 78 solutions from an institution one of us works at. We composed the first solution (Program 2a) by mixing a number of actual student solutions. Program 2b is an actual correct student solution (with variable names translated into English).

We ran three well-known static analysis tools on the three programs: PMD with the full set of rules, Checkstyle, and SonarLint[2] with default checks and full checks. Because Checkstyle always reported a subset of the PMD and/or SonarLint messages (besides layout), we omit Checkstyle messages from this paper.

3.2 Data analysis

We analysed the answers to the open question on which hints a teacher would give both qualitatively and quantitatively. We labelled the hint topics using an open coding method, and categorized

[1] The questionnaire can be found at www.hkeuning.nl/ImproveCode

[2] pmd.github.io/pmd-6.9.0, checkstyle.sourceforge.net (version 8.14), www.sonarlint.org/eclipse (version 4)

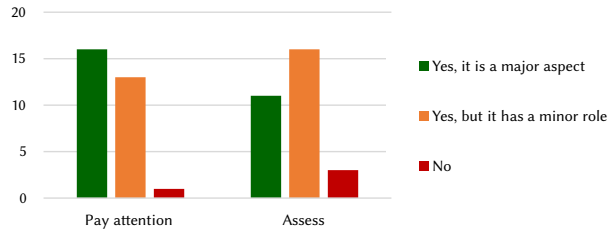

Figure 1: Responses to questions 'Do you pay attention to code quality while teaching programming to first- and second-year students?' and 'Do you explicitly assess/grade code quality aspects in programming assignments?'

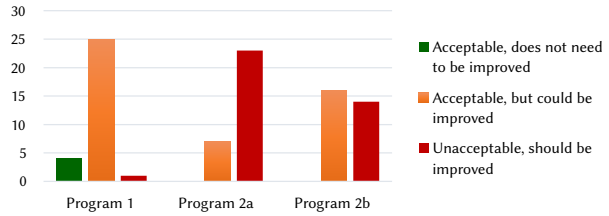

Figure 2: Responses to question 'How would you assess this solution in a formative situation (e.g. feedback during a lecture or lab)?' for all three programs.

them using the rubric by Stegeman et al. [18]. This rubric has been developed for assessing student code quality, based on a model with ten criteria. We assigned each topic to one of the four criteria that deal with algorithms and structure, which are *flow* (nesting, code duplication), *idiom* (choice of control structures, reusing library functions), *expressions* (complexity, suitability data types) and *decomposition*. The open coding was performed by one author. Another author checked the labelling of a randomly chosen 10% of the hints (with 89% agreement), and differences were discussed.

For the question on improving the program stepwise we performed a series of actions on the submitted programs. First, we removed steps in which the code was not changed (possibly a copy-paste issue). Next, because most participants probably did not use a compiler, we corrected syntax errors such as missing brackets and misspelled names, and converted to Java syntax (all given programs were written in Java, although we did not explicitly mention this). We also corrected some other small errors that were clearly unintentional. All programs were tested with a set of test cases. We assigned each program to a *cluster* based on the control flow of the program (loops, conditionals, branching and methods), because control flow shows the main structure and complexity of a method (the scope of the programs in our study). Clustering allows us to investigate similarities and differences without being distracted by details. We identified *transformations* as the add/edit/delete steps between two adjacent program *states* in a sequence.

4 RESULTS

4.1 Background of teachers

In total, 30 participants took part in our study. All participants teach programming and other CS-related courses in 3 different countries: The Netherlands (27), Sweden (3) and China (1). The 28 participants that reported their institute, teach at 15 different institutes: 10 in The Netherlands, 4 in Sweden and 1 in China, with between 1 and 5 teachers per institute. A few teach at more than one institute and country. The teachers have between 2 and 33 years of teaching experience, with an average of 11.4 years, and a median of 9 years. A total of 90% teach first year courses, 80% teach second year courses, and 70% teach courses for students in their third year or higher.

4.2 Role of code quality (RQ1)

We asked teachers if code quality appears in the learning goals of their first- and second-year programming courses, to which 23 replied with 'yes' and 7 with 'no'. Figure 1 shows to what extent teachers address and assess code quality aspects. Code quality clearly has a smaller role in assessment than in teaching.

A total of 11 out of 30 teachers do not advise or prescribe tools that deal with code quality/refactoring to their students. The 19 that do advise or prescribe tools, mostly mention static analysis tools (e.g. SonarQube, Checkstyle, linters) and IDE functionality or their plugins (e.g. Resharper). To a minor extent testing and code reviewing is mentioned. All tools mentioned are professional tools, and not explicitly intended for education.

4.3 Program hints and steps (RQ2 and RQ3)

4.3.1 Program 1. Running PMD on Program 1 reports that the for could be replaced by a foreach, and that unnecessary comparisons in boolean expressions should be avoided. SonarLint only reports on the equals true. Figure 2 shows how the teachers would assess this solution in a formative situation. Most teachers (25) answered 'acceptable, but could be improved'.

We asked the teachers to describe all hints they would give to a student to improve this program. Table 1 shows all hint topics and the number of mentions. For the issue that was pointed out the most, the flow inside the loop, some participants focussed more on the duplication, and others more on the complex if-structure.

Next, we asked the teachers how they would want the student to edit (refactor) the program step by step. In total 3 teachers did not provide any steps: 1 found the method specification itself problematic, the others did not give a reason. The remaining 27 teachers provided 2.8 steps on average, with a median of 3 (min 1, max 5) and a total of 76 program states, of which 11 were incorrect.

Regarding the type of loop, 10 teachers transformed the for into a foreach at various stages of the process, always as a single step in which nothing else was done. As a last step, 2 teachers replaced the loop by a functional style solution, calling a higher-level sum function on the array. Teachers mentioning this approach mostly said that they would only suggest it to more advanced students.

Regarding the flow in the loop, in 8 final programs the duplicated sum increment was still present. Removing the duplication by merging the ifs into a single statement, was often done as an early step, after removing the `==true`. In 2 cases a continue was used to skip a value that should not be added. However, we noticed merging the ifs was problematic: all 11 incorrect programs contained a merging mistake. Some of those mistakes were fixed in a next step.

We assigned each (intermediate) program to a cluster based on its control flow, identifying 13 clusters. The final programs were

Program 1: Most popular solution to exercise 1.

```
1  int sumValues(int [] values,
2                boolean positivesOnly) {
3      int sum = 0;
4      for (int i = 0;i < values.length;i++) {
5          if (positivesOnly == true) {
6              if (values[i] >= 0) {
7                  sum += values[i];
8              }
9          }
10         else {
11             sum += values[i];
12         }
13     }
14     return sum;
15 }
```

Table 1: Hint topics for program 1, as reported by 30 teachers. Indented topics are more specific.

Category	Description	Count
Expressions	Remove equals true (line 5)	20
	Do not add 0 to sum (line 6)	2
Flow	Improve flow in loop	1
	Improve nested ifs (line 5–12)	17
	Remove duplicated `sum +=` .. (line 7 and 11)	11
Idiom	Change type of loop (line 4)	11
	Use a higher-level function	3
Decomposition	Move common code to method	2
	`positivesOnly` check to method (line 5)	2
Other	General	2
	Misc. mentioned once (various categories)	5

Table 2: Correct end clusters for program 1.

Start:	12x:	3x:	2x:	6x:

`for`
`  if`
`  if`
`  else`
`return`
	`for/foreach`
	`  if`
	`return`
		`for/foreach`
		`  if`
		`  if`
		`  else`
		`return`
			`for/foreach`
			`  if`
			`  continue`
			`return`
				Others

distributed over 10 clusters; excluding 4 incorrect final programs, we counted 9 clusters, as shown in Table 2.

4.3.2 Program 2a. PMD reports three 'dataflow anomalies' for the `total` and `stop` variables in Program 2a. PMD considers this a low-priority issue that might not be problematic. PMD also points out equals false and the self-assignment. SonarLint with default settings also reports on equals false and even gives two messages on the self-assignment. With full checks, it mentions that the if-else if (lines 7–11) should end with an else, and a constant should be used for magic number 2.

The program contains a functional error regarding the stop condition. We instructed participants to ignore this error when answering the questions. However, the first 10 participants did not see this note and read that it was a correct solution. Figure 2 shows the response to the question on formatively assessing the solution, to which most teachers (23) answered it was unacceptable. The fact that the solution was incorrect could have contributed to this score.

Table 3 shows the hint topics for this program. Teachers often mentioned improving the complex flow in the loop, exiting from the loop when the stop condition was met, changing the loop type and removing clutter in the expressions.

Looking at the edit steps, 28 teachers provided 3.0 steps on average (min 1, max 5), with a median of 3 steps and a total of 83 program states. Two teachers did not provide any steps (they did not know what to suggest or what a student would have to do). Of the 83 program states, 8 were not functionally correct according to either stop condition (-1 or negative). We excluded 1 of the 28 sequences from the analyses below because it had unclear steps.

Although we did not explicitly ask this, 15 teachers tried (sometimes unsuccessfully) to fix the functional error. Most of them (11) started fixing in the first step; 3 of the others first removed clutter. In general, almost all clutter (the else with self-assignment, the redundant if) was removed by everyone mostly in the first or second step. The `==false` was mostly removed as part of another step.

The loop type was changed into a while 8 times at various stages of the process, and 1 teacher replaced the loop by recursion. Almost all participants changed the program to exit from the loop when the stop condition was met: 5 used a break, 5 used a return, and a majority of 14 added a stop condition to the loop header. Exiting from the loop was also done at various stages. The `stop` variable was eliminated from 20 sequences, and 7 kept it.

In total 20 clusters were identified, and the final programs were in 12 clusters. Table 4 shows the 10 final clusters if we exclude the 5 incorrect final programs.

4.3.3 Program 2b. Running PMD on the body of Program 2b reports multiple dataflow anomalies, and reports that the variable `number` could be made 'final'. SonarLint with full checks mentions magic number 2 on line 7. Figure 2 shows the formative assessment.

The hint topics for this program are shown in Table 5. The main topics were exiting from the loop when the stop condition is met, improving the complex flow in the loop, removing the duplicated increment, and replacing the foreach by another type of loop.

For the steps, 26 teachers provided 2.9 steps on average (min 1, max 5), with a median of 3 steps and a total of 76 program states. Of the 4 teachers that did not provide steps, 2 teachers advised that the student should start over instead of rewriting the program. The other 2 did not give a reason.

Teachers made multiple mistakes rewriting this program: of the 76 program states, 21 functionally incorrect programs were created by 10 teachers. The majority of the mistakes were related to incorrect indexing (not looping through only the odd indices). This mistake was often made in a step that also transformed the foreach into a for. We excluded 2 sequences from the analyses below because they had unclear steps, leaving 24 valid sequences.

Although in 2a the for-loop was mostly kept, followed by a few while-loops, in 2b we only counted 9 for-loops and 2 while-loops. The foreach was kept 13 times. Transforming the foreach into a for was mostly done in the first step, transforming to while was always done after a transformation into a for. Of the 11 that used a for or while, 6 skipped the even indices by incrementing the index

Program 2a: Solution to exercise 2.

```
1  int unevenSum(int [] array) {
2      int total = 0;
3      boolean stop = false;
4
5      for (int i = 1;i < array.length;i = i + 2) {
6          if (stop == false) {
7              if (array[i] >= 0) {
8                  total += array[i];
9              } else if (array[i] < 0) {
10                 stop = true;
11             }
12         }
13         else {
14             total = total;
15         }
16     }
17     return total;
18 }
```

Table 3: Hint topics for program 2a.

Category	Description	Count
Expressions	Remove self assignment (line 14)	12
	Remove equals false (line 6)	7
	Use compound operator += (line 5)	2
Flow	Exit from loop when done	20
	Improve flow in loop	6
	Remove redundant if (line 9)	10
	Remove unnecessary else (line 13–15)	7
	Reverse if-else (line 6–15)	2
Idiom	Change type of loop (line 5)	8
Other	Fix functional error stop condition	13
	General hints	9
	Add tests	3
	Misc. mentioned once (various categories)	7

by 2. For the remaining, 14 kept the modulo check, 2 introduced a boolean that switched between true and false, and 2 made a mistake.

Exiting from the loop was solved rather differently than in program 2a: more teachers used a break (10 vs. 5) and fewer teachers (6 vs. 14) added a stop condition to the loop header. This step was usually done somewhat later in the sequence.

All but 1 teacher removed the duplicated increment, mostly in an early step. Sometimes this was done in 2 steps: first moving the increment outside the if-else, followed by reversing the if-else and removing the then empty else. Transforming the foreach into a for also eliminates the duplication by moving the increment to the loop header.

Variables were renamed in a few cases: 4 renamed `answer` to `sum` or `total`, and `value` was occasionally renamed to `done` or `stop` if it was still used. Renaming was done at various stages.

In total we identified 31 clusters, and the final programs were in 14 clusters. Excluding 8 buggy final programs, we counted 11 clusters, as shown in Table 4. We would expect that the final programs after improving 2a match those of 2b (the error from 2a does not have to affect the cluster). However, only 5 participants ended in the same cluster for program 2a and 2b.

Table 4: Correct end clusters for program 2a and 2b.

Program 2a (n=22)

Start:

```
for
  if
    if
    else
      if
  else
return
```

6x:

```
for/while
return
```

4x:

```
for/while
  if
  else
return
```

2x:

```
for
  if
  else
    break
return
```

2x:

```
for
  if
    return
  else
return
```

2x:

```
for
  if
  else
    return
return
```

2x:

```
for
  if
    break
return
```

4x: Others

Program 2b (n=16)

Start:

```
for
  if
  else
    if
    if
return
```

4x:

```
foreach/for
  if
    if
      break
return
```

2x:

```
foreach/for
  if
    if
    if
return
```

2x:

```
for/while
return
```

8x: Others

5 DISCUSSION

In this section we answer the research questions and discuss the results in more depth.

RQ1. The results of RQ1 show that while code quality is certainly an important topic for most teachers, its role is smaller in the summative assessment of student code.

RQ2. Based on teacher feedback on three low-quality implementations of simple methods, the hints that teachers would give deal with improving control flow, choosing representative names, using suitable language constructs, removing clutter, and optimising the algorithm. Although some hint topics are mentioned by a majority, other topics are only mentioned by much smaller subgroups, implying that teachers do not consider the same things important. This finding contradicts Nutbrown and Higgins' claim that their assessors were in agreement on assessment criteria [14].

Regarding the form of the feedback, we noticed several aspects. Hints were often formulated as a question, such as 'Is there any code duplication that you could remove?' The amount of detail considering *why* something should be improved varied: for transforming a for into a foreach, an example with more motivation is: 'If we are traversing all values in the array, couldn't we use another type of loop?' Other hints mention that the 'other type of loop' is a foreach, and some even provide the syntax of the foreach header.

When comparing teacher hints to what professional tools report, we see many differences. Tools do not give feedback with increasing detail as teachers would do. Issues related to control flow and algorithmic optimisations are not pointed out by tools. A main reason is that tools do not know what the code they analyse should do. Also, default tool settings usually have high thresholds, so minor issues such as small duplicated blocks are usually not reported. Our

Program 2b: Actual student solution to exercise 2.

```
1  int unevenSum(int[] array) {
2      int answer = 0;
3      int index = 0;
4      boolean value = true;
5
6      for(int number: array) {
7          if(index % 2 == 0) {
8              index++;
9          } else {
10             if(number == -1) {
11                 value = false;
12             }
13             if(value) {
14                 answer = answer + number;
15             }
16             index++;
17         }
18     }
19     return answer;
20 }
```

findings support and complement the finding of Nutbrown and Higgins that static analysis tools miss context-specific issues [14].

RQ3. While others have already shown the great diversity in student solutions (e.g. [12]), our study also shows this diversity in teacher solutions. Remarkable is the difference in final states for program 2a and 2b that solve the same problem. Program 2b was probably the most problematic, and this starting point could have influenced the final program. For example, about half of the teachers kept the foreach in 2b, but a foreach is never introduced in 2a. Perhaps there are simply multiple ways that are equally fine, however, not all hints seem to be addressed in the final programs for program 2b. Finally, some teachers could have lost their focus for the last program, which would also explain the number of mistakes.

The large variety in steps and final programs makes it difficult to give an approach on how to improve a problematic program. General guidelines we can extract from the data are: remove clutter first, fix errors early, keep testing along the way (even teachers make mistakes), rename to meaningful names, and do larger refactorings later one step at a time. Starting over could sometimes be advisable. However, learning *why* a program has flaws, and *how* to address those flaws step by step could be a valuable learning experience. We provide some examples of stepwise improvement sequences with hints for the programs discussed in this paper online.[3]

If educators want to assess code quality, there should be agreement on what a high-quality program is. Code quality should not be a teacher's personal preference. One could argue that quality is not that important for novices, however, we argue that several issues pointed out in the previous section are caused by misunderstanding certain language constructs. Improving a solution may be a valuable way of learning more about how a programming language works. Although the issues mentioned in this paper are of a low level, and the topic of 'refactoring' is mostly associated with restructuring complex object-oriented software, we advise to start refactoring early with the statement/method level, to the class level later in a study program.

[3] www.hkeuning.nl/ImproveCode

Table 5: Hint topics for program 2b.

Category	Description	Count
Flow	Exit from loop when done	14
	Improve flow in loop (line 7–16)	5
	Remove `value` variable	3
	Reorder conditionals (line 7–16)	3
	Improve flow in else (line 10–16)	2
	Duplicated increment (line 8 and 16)	9
	Skip even indices	6
Idiom	Change type of loop (line 6)	14
Other	General hints	9
	Rename variable	7
	Misc. mentioned once (various categories)	9

5.1 Threats to validity

Because we only discussed three programs, we cannot generalise to all novice programs. However, the programs cover a broad set of constructs, and in this study we particularly aimed to make code quality, a potentially vague topic, more concrete by working with actual code. Conducting interviews could give us higher-quality data, but would not have given us insight into the diversity of the responses. It would be an interesting next step to discuss the various responses with the teachers to arrive at a more general view of which hints to give and which steps to take.

This study does not consider the responses of students to hints. A teacher would possibly adapt and give a more concrete hint when a student does not understand the initial hint. This was even mentioned by some participants in their responses. There might also be some differences between the type of hints for absolute beginners and second-year students.

6 CONCLUSION AND FUTURE WORK

This paper describes a study in which we asked teachers for their opinion on the quality of student code and how they would help students to improve it. While teachers find the topic of code quality important, they have different views on how to improve code. Teachers mostly agree on issues related to reducing algorithmic complexity and removing clutter, but they give different subsets of hints. Professional code quality tools do not point out these algorithmic complexity topics that teachers mention. We also discussed the great diversity in the final programs, which is influenced by the initial state of the code, and derived some general guidelines in how to approach an improvement sequence.

In future work we intend to use our findings to build better tools that help students improve the quality of their code. This research also calls for more debate on what a high-quality solution would look like for a novice. Experiments in the classroom with students are required to further study how we should learn students to improve their code, to which this study contributes.

ACKNOWLEDGMENTS

This research is supported by the Netherlands Organisation for Scientific Research (NWO), grant number 023.005.063.

REFERENCES

[1] Shamsa Abid, Hamid Abdul Basit, and Naveed Arshad. 2015. Reflections on Teaching Refactoring: A Tale of Two Projects. In *Proceedings of ITiCSE*. 225–230. https://doi.org/10.1145/2729094.2742617

[2] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research* 3 (2004), 245–262.

[3] Hannah Blau and J. Eliot B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proceedings of ITiCSE*. 15–20. https://doi.org/10.1145/2729094.2742622

[4] Jürgen Börstler, Marie Nordström, and James H Paterson. 2011. On the quality of examples in introductory Java textbooks. *ACM Transactions on Computing Education (TOCE)* 11, 1 (2011), 1–21. https://doi.org/10.1145/1921607.1921610

[5] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2017. "I know it when I see it" Perceptions of Code Quality. In *Proceedings of ITiCSE, Working Group Reports*. 70–85. https://doi.org/10.1145/3174781.3174785

[6] Dennis Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. In *Proceedings of ITiCSE*. 13–17. https://doi.org/10.1145/1999747.1999754

[7] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)* 17, 2 (2017), 7. https://doi.org/10.1145/2994154

[8] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding Semantic Style by Analysing Student Code. In *Proceedings of the Australasian Computing Education Conference*. 73–82. https://doi.org/10.1145/3160489.3160500

[9] Serge Demeyer, Filip Van Rysselberghe, Tudor Girba, Jacek Ratzinger, Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza, et al. 2005. The LAN-simulation: a refactoring teaching example. In *International Workshop on Principles of Software Evolution*. IEEE, 123–131. https://doi.org/10.1109/IWPSE.2005.30

[10] Martin Fowler. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[11] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of ITiCSE*. 110–115. https://doi.org/10.1145/3059009.3059061

[12] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the Differences Between Correct Student Solutions. In *Proceedings of ITiCSE*. 177–182. https://doi.org/10.1145/2462476.2462505

[13] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition.* Microsoft Press.

[14] Stephen Nutbrown and Colin Higgins. 2016. Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education* 26, 2-3 (2016), 104–128. https://doi.org/10.1080/08993408.2016.1179865

[15] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *Proceedings of SIGCSE*. 410–415. https://doi.org/10.1145/2676723.2677279

[16] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education (TOCE)* 18, 1, Article 1 (2017), 1:1–1:24 pages. https://doi.org/10.1145/3077618

[17] Suzanne Smith, Sara Stoecklin, and Catharina Serino. 2006. An Innovative Approach to Teaching Refactoring. In *Proceedings of SIGCSE*. 349–353. https://doi.org/10.1145/1121341.1121451

[18] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the Koli Calling International Conference on Computing Education Research*. ACM, 160–164. https://doi.org/10.1145/2999541.2999555

[19] Sara Stoecklin, Suzanne Smith, and Catharina Serino. 2007. Teaching Students to Build Well Formed Object-oriented Methods Through Refactoring. In *Proceedings of SIGCSE*. 145–149. https://doi.org/10.1145/1227310.1227364

[20] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *ACM Conference on Learning @ Scale*. 41–50. https://doi.org/10.1145/3051457.3051469