How teachers would help students to improve their code

Citation for published version (APA):

DOI:
10.1145/3304221.3319780

Document status and date:
Published: 02/07/2019

Document Version:
Publisher's PDF, also known as Version of record

Document license:
Taverne

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:
https://www.ou.nl/taverne-agreement

Take down policy
If you believe that this document breaches copyright please contact us at:
pure-support@ou.nl
providing details and we will investigate your claim.
How Teachers Would Help Students to Improve Their Code

Hieke Keuning
Open University of the Netherlands
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

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 questionnaire1 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 SonarLint2 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

---

1The questionnaire can be found at www.hkunming.nl/ImproveCode
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.

```java
int sumValues(int[] values, boolean positivesOnly) {
    int sum = 0;
    for (int i = 0; i < values.length; i++) {
        if (positivesOnly == true) {
            if (values[i] >= 0) {
                sum += values[i];
            }
        } else {
            sum += values[i];
        }
    }
    return sum;
}
```

Table 2: Correct end clusters for program 1.

<table>
<thead>
<tr>
<th>Start</th>
<th>12x:</th>
<th>3x:</th>
<th>2x:</th>
<th>6x:</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>if</td>
<td>else</td>
<td>return</td>
<td>for</td>
<td>foreach</td>
</tr>
<tr>
<td>if</td>
<td></td>
<td></td>
<td></td>
<td>for</td>
<td>foreach</td>
</tr>
<tr>
<td>return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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.

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.

```java
public int unevenSum(int[] array) {
    int total = 0;
    boolean stop = false;
    for (int i = 1; i < array.length; i = i + 2) {
        if (stop == false) {
            if (array[i] >= 0) {
                total += array[i];
            } else if (array[i] < 0) {
                stop = true;
            }
        } else {
            total = total;
        } else {
            return total;
        }
    }
    return total;
}
```

Table 3: Hint topics for program 2a.

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

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

<table>
<thead>
<tr>
<th>Program 2a (n=22)</th>
<th>6x:</th>
<th>4x:</th>
<th>2x:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ```java
for
if
else
else
return
``` |     |     |     |
| **4x:**            |     |     |     |
| ```java
for/while
if
else
if
else
return
``` |     |     |     |
| **2x:**            |     |     |     |
| ```java
for
if
else
else
return
``` |     |     |     |
| **2x:**            |     |     |     |
| ```java
for
if
else
else
return
``` |     |     |     |
| **4x:**            |     |     |     |
| ```java
for
if
else
else
break
return
``` |     |     |     |

<table>
<thead>
<tr>
<th>Program 2b (n=16)</th>
<th>6x:</th>
<th>2x:</th>
<th>8x:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ```java
for
if
else
if
return
``` |     |     |     |
| **4x:**            |     |     |     |
| ```java
foreach/for
if
if
if
break
return
``` |     |     |     |
| **2x:**            |     |     |     |
| ```java
foreach/for
if
if
if
if
return
``` |     |     |     |
| **8x:**            |     |     |     |
| ```java
for/while
if
if
if
if
return
``` |     |     |     |

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
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.

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.

---

3www.hkeuning.nl/ImproveCode
REFERENCES


