A Tutoring System to Learn Code Refactoring

Hieke Keuning
Open University of the Netherlands
and Utrecht University
h.w.keuning@uu.nl

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

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

ABSTRACT

In the last few decades, numerous tutoring systems and assessment tools have been developed to support students with learning programming, giving hints on correcting errors, showing which test cases do not succeed, and grading their overall solutions. The focus has been less on helping students write code with good style and quality. There are several professional tools that can help, but they are not targeted at novice programmers.

This paper describes a tutoring system that lets students practice with improving small programs that are already functionally correct. The system is based on rules that are extracted from input by teachers collected in a preliminary study, a subset of rules taken from professional tools, and other literature. Rules define how a code construct can be rewritten into a better variant, without changing its functionality. Rules can be combined to form rewrite strategies, similar to refactorings offered by most IDEs. The student can ask for hints and feedback at each step.

We describe the design of the system, show example sessions, and evaluate and discuss its contribution and limitations.

CCS CONCEPTS

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

KEYWORDS

Learning programming, tutoring systems, code quality, refactoring

ACM Reference Format:
Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A Tutoring System to Learn Code Refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432526

1 INTRODUCTION

Student misconceptions have always had much attention in studies on student programming [9, 28]. The focus has been mostly on programming mistakes resulting in functionally incorrect code. At the same time, there may be numerous functionally correct solutions to the same programming exercise [23], which are not always equally good. In 2017, an ITiCSE working group investigated how professionals, educators and students perceive code *quality*, finding a great diversity in its definition [7]. Their study also concluded that the topic of code quality is underrepresented in education. Recently, there has been increased attention to the style and quality of student solutions. Poor coding style and quality may lead to incomprehensible code that has low maintainability and testability, which is an issue even for professional software developers, not to mention novices. While one might argue that novice programmers should not be bothered too much with style and quality, these quality issues might point at underlying misconceptions. Also, *refactoring* code is an important skill that every programmer should possess, and novices are usually confronted early with code analysis and refactoring tools, which are increasingly a part of modern IDEs.

Teachers play an important part in how critically students view their code, but large class sizes prevent them from giving personalised feedback on student solutions. In a previous study, we have investigated how teachers would help students to improve their code [19]. We showed experienced teachers a number of functionally correct, but imperfect student solutions and asked them which hints they would give and how they would want a student to refactor code. We compared the teacher hints with the output of professional static code analysis tools, and concluded that these tools are not suitable for giving meaningful feedback to novices.

This paper describes a *tutoring system* to complement human tutoring, giving hints and feedback on exercises in which students improve code. The contributions are (1) the design of a tutoring system that helps students learn about code improvement and better suits the requirements for novice programmers, and (2) its validation based on a technical evaluation, and student use. The system is available online[1].

Section 2 provides background and describes related tools. Section 3 shows an example session. Section 4 shows the design, which is evaluated and discussed in Section 5. Section 6 concludes and describes future work.

2 BACKGROUND AND RELATED WORK

This section provides some background on code quality and code refactoring, and discusses professional tools and their use in education, as well as tutoring systems specifically intended for education.

2.1 Code quality and refactoring

The aim of our system is to teach students about code quality in the context of small programs, which are mostly single methods. The definition of code quality we employ revolves around the directly observable properties of source code, such as algorithmic aspects

[1] www.hkeuning.nl/rpt

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432526

(flow, expressions, language constructs) and structure (decomposition, modularization). Layout and commenting are also relevant, but are beyond our research scope because they are not that complex, and existing tools are often good enough to support students.

Several terms are used to indicate problems with code quality, such as flaws, issues, violations and the well-known code *smells* introduced by Fowler [15]. Code smells are characteristics in code that might point at a problem with the design of the code, although it is functionally correct. These problems can have an impact on quality attributes such as maintainability, performance and security.

Code *refactoring* is improving code step by step while preserving its functionality. Fowler [15] provides a collection of refactorings, mainly focused on code structure. Code Complete [24], a well-known software construction handbook, describes refactorings on multiple levels: data, statement, routine, class implementation, class interface and system. We focus on data-, statement- and routine-level refactorings, which are most relevant for beginner programs. Examples of such issues include code duplication, overly complex or unnecessary constructs, and unsuitable language constructs.

Multiple studies have investigated the presence of flaws in student code that are not functional errors [8, 12, 13, 18, 27]. These studies show that flaws are abundantly present, and there is not a great deal of improvement for certain issues. Because studies show varying results, fixing issues and preventing them in future tasks seems to be highly dependent upon the context. There is also some evidence that the presence of flaws that may point at actual bugs during the process correlates with submitting incorrect code [13].

2.2 Professional tools

Relevant professional tools are either static analysis tools or refactoring systems. Both are often integrated in IDEs; static analysis tools are usually also available as a stand-alone tool. Static analysis tools automatically detect quality issues and code smells in code, and generate a list of issues as output, which are usually violated rules. Examples are FindBugs, Checkstyle, PMD, SonarQube, Resharper, and linters. Several IDEs offer support for refactoring code, either as integrated functionality or as an extension that can be installed. Some examples are Visual Studio, Eclipse, and IntelliJ. A 2012 study showed that refactoring tools are being used infrequently by professional programmers, and that programmers perform quite a lot of low-level refactorings (at the block-level) [25].

Some research exists on the use of professional tools in education. Nutbrown and Higgins [26] have studied whether static analysis tools can be used for summative assessment of student programs. The authors designed a grading mechanism based on the ratings of PMD rules, and compared the automated grades to the grades of instructors. They conclude that the correlation was not strong enough and some manual assessment was still needed, in particular for context-specific issues. Edwards et al. [14] explored whether the FindBugs tool can be used to help struggling students, and found a subset of tool warnings that correlate with incorrect code. However, the authors have not used the tool with students yet.

2.3 Tutoring systems

A systematic literature review of tools that generate automated feedback for programming exercises shows there has been a lot of work focussing on the mistakes that students make, but less work on the style and quality of student programs [21]. The study also found that there is much more emphasis on assessment than on guidance to help students improve their programs. Many of these tools are automated assessment tools, which are usually more focussed on grading finished programs. Another type of tool is the Intelligent Tutoring System (ITS), which helps students by guiding them step by step towards a solution [31]. VanLehn [32] found in his experiments that ITSs were nearly as effective as human tutoring. Several ITSs exist for the programming domain [11], offering adaptive feedback (the 'inner loop'), navigational support (the 'outer loop') and several additional features such as programming plan support, reference materials and worked examples.

There are also some tools designed specifically for education that analyse code quality. FrenchPress [6] is a plugin that reports student-friendly messages for a small set of programming flaws. Style++ generates a report with style issues such as commenting, naming and code size [5]. WebTA [30] is a programming environment that reports on failed tests, common errors, and also more stylistic issues. AutoStyle [10] gives stepwise feedback on how to improve the style of correct programs. An experiment with students using AutoStyle has shown improvements, but students also still struggled with improving style [34]. AutoStyle is different from our tool because it relies on historical student data.

2.4 Teachers' perspective and conclusion

We recently conducted a study with 30 experienced CS teachers, investigating how they address code quality in general, and having them assess student code of low quality [19]. We asked which hints they would give and how the student should improve the code step by step. We compared their suggestions to the output of PMD, Checkstyle and SonarQube. Based on these findings, other literature, and some of our own observations, we generally consider professional tools in their current form problematic for novice programmers. We summarize the reasons:

i) The terminology and phrasing of messages can be too hard to understand by novices.
ii) All issues are reported at once, which may overwhelm the student and cause cognitive overload.
iii) Not all reported issues are relevant for novices.
iv) Because these tools do not know what the programmer is working on, feedback is not tailored to the current task and its requirements, and the level of the student.
v) IDEs execute a refactoring in a single step, which does not give novices much insight into how a refactoring works.
vi) IDEs may offer code changes that are *possible*, but not necessarily *useful*.

3 A TUTORING SESSION

In this section we demonstrate how our system works by showing two tutoring sessions for different exercises. The target audience are students in higher education who already know the basics of programming (control structures, loops, arrays, methods, etc.), who would typically be CS majors. The system offers exercises in which an exercise specification and a functionally correct, but inelegant program is given. It is the student's task to improve (refactor) the

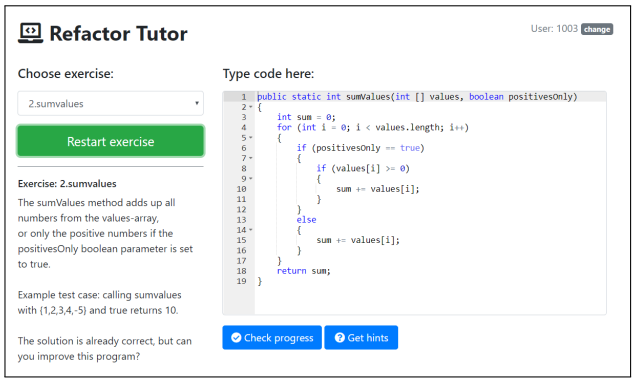

Figure 1: Web application for the tutoring system.

program to make it more elegant/efficient/readable. Currently, there are six exercises, and new similar exercises can be added easily.

Fig. 1 shows a screenshot of the web interface of the system. For editing code we used the open source Ace editor [3], which supports syntax highlighting, automatic indentation, highlighting matching parentheses, code folding and more. A student has two ways to ask for feedback during programming: CHECK PROGRESS and GET HINTS. The CHECK PROGRESS button checks the current state of the program and reports on mistakes (syntax errors, failed test cases, known incorrect steps) or successful steps. The system presents hints in a tree structure, of which the first option is shown by default. The student can click on EXPLAIN MORE (denoted by the ↳ symbol) to get a more detailed hint, or click on ANOTHER HINT (denoted by the ⤵ symbol). In the examples we show parts of the hint tree and fold certain branches for clarity (the ↳ and ⤵ symbols indicate there are more hidden hints), and we only show affected code fragments, omitting details and highlighting the major changes. We show the output as text in a sans-serif font.

3.1 Example 1: Sum of values

The first exercise is taken from another study [23], with the most popular student solution as the starting program.

Tutor. The sumValues method adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true. The solution is already correct, but can you improve this program?

```
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;
}
```

Student. The student asks for a hint right away.

Tutor. The tutor responds with a tree of hints that the student expands step by step by clicking on the ↳ icon.

- Can you simplify a boolean expression? ⤵
  - == true may be removed.
    - Try to use this example code: `positivesOnly`

Student. The student removes the equals true and asks the tutor to check her step, and then asks for a new hint.

```
    for (int i = 0;i < values.length;i++) {
        if (positivesOnly) {
            ..
        }
        ..
    }
```

Tutor. That was a correct step, well done!

- Can you simplify the condition in the if? ⤵

Student. The student notices that when `values[i]` contains 0, addition has no effect.

```
            if (values[i] > 0) {
                sum += values[i];
            }
```

Tutor. That was a correct step, well done!

- There is some duplication, can you simplify the if-statements to remove it? ⤵
  - Can you combine the conditions, so you only need 1 if? ↳

Student. The student incorrectly combines the conditions.

```
        if (! positivesOnly && values[i] > 0) {
                sum += values[i];
        }
```

Tutor. The tutor recognises this step as an incorrect attempt to combine two conditions.

Student. The student expands the last hint, uses the code example for the first step and asks for the next hint.

```
if((positivesOnly && values[i] > 0) || !positivesOnly)
        sum += values[i];
```

Tutor. Can you shorten the expression between the brackets?

```
if (values[i] > 0 || !positivesOnly)
    sum += values[i];
```

Tutor.

- Can you replace the for-loop by a foreach-loop? ↳

Student. The student rewrites the for-loop.

```
    for (int i: values) {
        if (i > 0 || !positivesOnly)
            sum += i;
    }
```

Tutor. Well done, no more improvements left.

3.2 Example 2: Odd sum

The starting program for this exercise, taken from our previous study [19], is a mix of a number of student solutions collected at the university one of us works at.

Tutor. The method oddSum returns the sum of the numbers at odd indices in the array parameter, until the number -1 is seen at an odd index. The solution is already correct, but can you improve this program?

```
1  int oddSum(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] != -1) {
8                  total += array[i];
9              } else if (array[i] == -1) {
10                 stop = true;
11             }
12         }
13         else {
14             total = total;
15         }
16     }
17     return total;
18 }
```

Student. The student explores her options by clicking on alternative ⤵, and makes three edits.

Tutor.

- Do you know a shorter way to write a calculation? ↳
- You're assigning a variable to itself, is that useful? ↳
- Can you simplify a boolean expression? ↳ ⤵

```
1      for (int i = 1;i < array.length;i += 2) {
2          if (!stop) {
3              ..
4          }
5          // removed the else-block
6      }
```

Tutor. All test cases still pass.

- Can you find an else-if condition that is not necessary?
  - We don't need the check in the else-if, because we know it will be true. ↳
- Can we immediately exit from the loop once we're done?

```
1              if (array[i] != -1) {
2                  total += array[i];
3              } else {
4                  stop = true;
5              }
6          }
```

Student. The student removes the if.
We omit the rest of this session.

4 DESIGN

This section describes a technical view of the design of the tutoring system. The system has been developed using a 'design science' approach, described by Wieringa [33] as 'design and investigation of artifacts in a context'. A design cycle is composed of problem investigation, treatment design and treatment validation, and is part of a larger engineering cycle, in which the treatment is implemented and evaluated in the real world. A design cycle is typically executed multiple times in a project. This paper focuses on the initial design cycle, for which problem investigation has mostly been done in our preliminary study (see Section 2.4).

The tutor supports refactoring strategies consisting of multiple steps, which transform an imperfect solution into an improved solution. Appropriate feedback messages are attached to the steps in the strategy. The tutor supports exercises of class 3 according to the classification of Le and Pinkwart [22], implying a student can follow multiple solution strategies to solve an exercise. To add a new exercise, the following elements have to be provided:

- A file with the starting code to be refactored.
- A text file describing the exercise.
- A set of test cases, consisting of input/output pairs. These are currently hard-coded, but should be provided in a separate file in the near future.

The supported programming language is a subset of Java that includes assignments, arithmetic/comparison/boolean operators, several primitive data types (ints, bools, strings, doubles), arrays, branching, loop statements, and methods. The system uses an internal data type of a fairly generic object-oriented language, so using another language would require translation to that data type. We expect this to be feasible, but have not attempted this yet.

Section 4.1 describes the architecture. Sections 4.2 and 4.3 focus on the implementation of the domain knowledge, and Section 4.4 on how the system generates its output by providing feedback *services*.

4.1 Architecture

The system consists of a web-based interface and a backend that processes JSON requests and replies with JSON responses. This design makes it possible to create a different user interface or (IDE) plugin, which uses the hint and feedback services from the existing backend. The backend calls the Oracle Java compiler to retrieve compiler error messages. All requests and responses are logged in a database. The current state of the code is attached to these requests.

4.2 Technology

We have developed our tutoring system on top of the Ideas framework [4] for developing ITSs. Tutors built with Ideas (Interactive domain-specific exercise assistants) can provide stepwise automated hints for exercises in various domains, such as mathematics and programming [17]. Various feedback services are offered, such as next-step hints, validation of steps, and showing complete solution paths. *Rules* and *strategies* have to be specified to provide these services. Rules are transformations on the data type of the domain, such as refining or rewriting (parts of) a student program. In the refactoring context, a simple example of such a rule is rewriting `x==true` into `x` (more in Section 4.3). Each rule or refactoring should preserve the functionality of the program.

We use *normalisations* to transform (parts of) a program to a *normal form*, by applying a large set of rewrite rules that are not

necessarily refactorings, such as changing the order of a calculation (y+1+x ⇝ 1+x+y) and removing syntactic sugar (x+=2 ⇝ x=x+2). Normalisation has been used before in programming tutors [16, 29, 35], mainly to recognise more variants of the same program. In the context of refactoring, it also simplifies the definition and implementation of refactorings, because fewer variants have to be considered. To generate feedback specific for a student's implementation, normalisations have to be 'undone' [29], which we currently have not implemented, causing the hints with code examples to not exactly match with the student's code at times.

4.3 Rules and strategies

The rules are the main building blocks of the tutoring system. We have based the rules on several sources from the literature as well as best practices from software engineering:

- Rewrite steps suggested by teachers, identified in our previous study [19].
- Semantic Style Indicators (SSIs) identified in student programs by DeRuvo et al. [12]. An SSI is defined as 'a pattern of a short sequence of statements that in some circumstances could be considered sub-optimal'. Currently 10 of their 16 SSIs are implemented in our system.
- Semantics-preserving variations (SPVs) that occur in student programs, of which Xu and Chee [35] distinguish 13 types. An SPV changes the computational behaviour of a program while preserving computational results. We account for several of these variations in our rules and other normalisations.
- Rules from professional tools, in particular the code suggestions from PMD [1], a static analysis tool mostly used for Java, and IntelliJ [2], a Java IDE that provides many code analysis and refactoring options.
- Equality rules from arithmetic and logic (e.g. absorption or identity operations).

Table 1 summarises the rules currently in our system. The columns TCH (teacher hints and steps), SSI ([12]), PT (professional tools) and A/L (arithmetic and logic rules) indicate the source of the rules.

It is important to consider the *soundness* and *completeness* of our rule set. Although we do not strive for completeness, because it is impossible to foresee every single solution to a programming problem, we want to minimise cases in which our system tells the student there are no hints left, while their solution is imperfect. Although we do not formally prove soundness, we believe the rules are sound because they represent mathematical and logical rules, and adhere to the semantics of programming language constructs. Continually checking a program against a set of test cases also ensures that the behaviour of the program is preserved.

An example of a simple rule is removal of a useless condition:

```
if (c) a; else if (!c) b; ⇝ if (c) a; else b;
```

If the pattern on the left of the arrow is detected, the statement can be rewritten into the statement on the right. More complex rules, such as the transformation of a for-loop into a foreach, require verifying several conditions in advance, such as checking that all array values are addressed from first to last, and that the array is not being modified inside the loop. If all conditions are met, the corresponding foreach will be generated.

Table 1: Rule summary. Entries denoted by * have several accompanying rules.

Description	TCH	SSI	PT	A/L
Expressions				
Simplify boolean expressions*	●		●	●
Optimise calculation	●			
Improve odd/even check				●
Use compound, incr., decr. operators*	●		●	
Remove self-assignment	●	●	●	
Branching				
Simplify by removing duplication/nesting*	●	●		
Extract duplicate statement from if/else*		●		
Remove redundant conditional check	●	●		
Remove empty or useless if/if-else/else*	●	●	●	
Reverse negative if-else			●	
Loops				
Change for-loop into foreach	●		●	
Change for-loop into while	●		●	
Exit loop when done with condition or break*	●			
Replace loop by calculation				●
Remove break from loop				
Statements				
Remove empty statements*			●	
Simplify if/else returning bools by single return*		●		
Buggy				
Incorrect collapsing of if/else*	●			
Incorrect equals true replacement				

The Ideas framework also supports the definition of *buggy rules*, which describe invalid transformations that change the computational semantics of code. For now we have only implemented a small set of these buggy rules. An example of a simple buggy rule is the incorrect disjunction of the two if conditions c1 and c2:

```
if (c1) if (c2) a; ⇝ if (c1 || c2) a;
```

Rules are combined to define more elaborate *strategies*, which describe the step-by-step solution to a problem. In strategies, rules can be combined in sequences, chosen as (prioritised) options, and navigation rules can traverse the abstract syntax tree to apply rules at specific locations. From our teacher input we derived that teachers advise to clean-up code first before moving on to more complex refactorings; we implemented this in the strategy by enforcing cleanup rules before enabling certain other rules.

4.4 Feedback services

The system offers the following feedback services:

Hint tree. Hints are generated by calculating the first possible steps of the strategy. The hint tree contains all available hints in a hierarchical structure, as described in Section 3. A feedback script is used to store the hint messages attached to each step. The script contains key-value pairs that can easily be adjusted by a teacher. The example below shows the feedback messages for two levels of hints for the same issue. The third level of feedback is usually a code fragment demonstrating the refactoring.

```
feedback removeUselessIfS  = Do we need the if-statement?
feedback removeUselessIf.1 = The condition in the if is always
                             true, so we don't need the if
```

Hints remaining. The current number of top-level hints.

Diagnosis. This function checks the current state of the student program. If the program cannot be parsed, it produces an error message. Next, it tries to recognise if a buggy rule has been applied. If not, it uses test cases to verify that the program still has the required functional behaviour. If it detects that a known rule has been correctly applied, it reports that the student just successfully applied that rule. If multiple rules have been applied, or unknown edits have been done, it just reports that the program is correct.

5 EVALUATION AND DISCUSSION

We provide a technical evaluation, summarise the results of an evaluation with students, and discuss results and limitations.

5.1 Teacher data evaluation

Our technical evaluation checks if edits to program states are recognised. We use the two exercises from the example tutoring session, for which we also have teacher data: SumValues and OddSum. In [19] we asked teachers how they would want a student to refactor these programs step by step. We analysed data of 27 teachers who provided 76 new program states (excluding the start state) for SumValues. We excluded 11 functionally incorrect programs that our tutor rightly identified, 5 programs that used language constructs our tutor does not support, and 3 other invalid states. We let our system generate all available hints for the remaining 57 programs: for 43 programs the teacher's edits for that step were in this hint set (75.4%), for 3 some edits were in the set but some were not (5.3%), and for 11 none of the edits were in the set (19.3%).

For the OddSum exercise 27 teachers provided 66 valid program states. We found that for 41 programs the teacher's edits were in the hint set (62.1%), and for 16 the edits were partially in the set (24.2%). We noticed that some teachers solved some issues differently from the concrete hints the system gave, but we mark these edits as successful because the hint does not appear any more.

We can conclude that the hints generated for the majority of the states lead to what the teacher would suggest to do next. Usually multiple hints are available for a state (even for final states), allowing for the various solution paths we saw in the teacher data.

5.2 Student evaluation

Because this paper is primarily a software report describing a tool, we have focussed on the functional and technical design of the system. However, we have recently conducted a study with 133 students using the system. The log data and student evaluations show that the hints help students to solve refactoring exercises, and that students request hints at various levels, regularly check their solutions against the test cases, and value working with the system. We have also derived several improvements from this analysis to be incorporated in the next cycle of our design science process. We provide a detailed analysis elsewhere [20].

5.3 Discussion

In this section we summarise and discuss how our tool attempts to solve the problems listed in Section 2.4:

i) Terminology and phrasing are targeted at novices, and can be adjusted by teachers. Most high-level hints are phrased as questions, which teachers often did as well.
ii) Issues can be shown gradually by letting the student ask to make a hint more specific, or to request a different hint.
iii) We have selected a subset of issues relevant for novices. In future work teachers should be able to switch off rules they may find unsuitable for a particular group of students or course.
iv) Although the issues we support go beyond what professional tools detect, we consider exercise- and student-specific feedback to be future work.
v) Our system can guide a student through more complex refactorings step by step.
vi) Our system does not offer edits with no particular goal, instead we offer edits based on the input of experienced teachers.

The proposed system is a practice tool encouraging students to critically assess code and think of alternative solutions. It provides an opportunity to explore other language constructs, and more carefully consider control flow and structure. The hints are suggestions that should trigger further discussion among teachers, and between teachers and students. Although novices often produce verbose code, because they might find it easy to understand (or perhaps it just worked), at a certain time they should move beyond that. We therefore advice the system to be used by students with some programming experience who are ready for the next step.

5.4 Limitations

The system currently contains six exercises with accompanying rules, which may raise questions on the generalisability of the feedback mechanism of the system. However, the majority of the rules are not specific to an exercise, and can be reused for other exercises as well. We do need to expand the set of rules, and also implement a more dynamic way of devising rules, which is described as future work in Section 6. Studies analysing how students use the tutoring system, and how they learn from it, should give us more insight into the effectiveness of the system.

6 CONCLUSION AND FUTURE WORK

This paper describes the functionality and design of a tutoring system that teaches students about improving code. We have shown a tutoring session in which students receive hints on how to improve an already correct piece of code, and get feedback on the correctness of their steps. We have shown that the behaviour of the tutor matches with how teachers want students to improve their code. We also show that the tutor goes beyond what professional static analysis tools and IDEs do, and better meets the needs of students. We have obtained encouraging results from our exploratory study of students using the system to solve refactoring exercises.

As part of our design science process, we will iteratively improve the system based on the findings. We plan to add new features, such as having the teacher provide model solutions, from which additional improvement rules can be extracted and dynamically used in the system. Also, we want to add more buggy rules. Future experiments could compare the effects of using the tutoring system to those of professional tools, and study the effect on student code quality in the long run.

ACKNOWLEDGMENTS

This research is supported by the Dutch Research Council (NWO), grant number 023.005.063.

REFERENCES

[1] 2018. PMD. (2018). https://pmd.github.io/pmd-6.9.0/
[2] 2019. IntelliJ IDEA Community Edition 2019.2. (2019). https://www.jetbrains.com/idea
[3] 2020. Ace editor. (2020). https://ace.c9.io/
[4] 2020. Ideas. (2020). http://hackage.haskell.org/package/ideas
[5] 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.
[6] Hannah Blau and J. Eliot B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proceedings of ITiCSE*. 15–20. DOI: https://doi.org/10.1145/2729094.2742622
[7] 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. DOI: https://doi.org/10.1145/3174781.3174785
[8] Dennis Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. In *Proceedings of ITiCSE*. 13–17. DOI: https://doi.org/10.1145/1999747.1999754
[9] 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. DOI: https://doi.org/10.1145/2994154
[10] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-Driven Automatic Hint Generation for Coding Style. In *International Conference on Intelligent Tutoring Systems*. Vol. 9684 LNCS. 122–132. DOI: https://doi.org/10.1007/978-3-319-39583-8_12
[11] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*. ACM, 53–62. DOI: https://doi.org/10.1145/3160489.3160492
[12] 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. DOI: https://doi.org/10.1145/3160489.3160500
[13] Stephen Edwards, Nischel Kandru, and Mukund Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 65–73. DOI: https://doi.org/10.1145/3105726.3106182
[14] Stephen Edwards, Jaime Spacco, and David Hovemeyer. 2019. Can Industrial-Strength Static Analysis Be Used to Help Students Who Are Struggling to Complete Programming Activities?. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 7825–7834. DOI: https://doi.org/10.24251/HICSS.2019.941
[15] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[16] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Thomas van Binsbergen. 2017. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 65–100.
[17] B. Heeren and J. Jeuring. 2014. Feedback services for stepwise exercises. *Science of Computer Programming* 88 (2014), 110–129. DOI: https://doi.org/10.1016/j.scico.2014.02.021
[18] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *ITiCSE*. 110–115. DOI: https://doi.org/10.1145/3059009.3059061
[19] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How Teachers Would Help Students to Improve Their Code. In *ITiCSE*. 119–125. DOI: https://doi.org/10.1145/3304221.3319780
[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student Refactoring Behaviour in a Programming Tutor. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. DOI: https://doi.org/10.1145/3428029.3428043
[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018). DOI: https://doi.org/10.1145/3231711
[22] Nguyen-Thinh Le and Niels Pinkwart. 2014. Towards a Classification for Programming Exercises. In *Workshop on AI-supported Education for Computer Science*. 51–60.
[23] 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. DOI: https://doi.org/10.1145/2462476.2462505
[24] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press.
[25] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18. DOI: https://doi.org/10.1109/TSE.2011.41
[26] 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. DOI: https://doi.org/10.1080/08993408.2016.1179865
[27] 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. DOI: https://doi.org/10.1145/2676723.2677279
[28] 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. DOI: https://doi.org/10.1145/3077618
[29] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64. DOI: https://doi.org/10.1007/s40593-015-0070-z
[30] Leo C. Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. In *Proceedings of SIGCSE*. ACM, 738âĂŞ744. DOI: https://doi.org/10.1145/3287324.3287463
[31] Kurt VanLehn. 2006. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education* 16, 3 (2006), 227–265.
[32] Kurt VanLehn. 2011. The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. *Educational Psychologist* 46, 4 (2011), 197–221. DOI: https://doi.org/10.1080/00461520.2011.611369
[33] Roel Wieringa. 2014. *Design science methodology for information systems and software engineering*. Springer. DOI: https://doi.org/10.1007/978-3-662-43839-8
[34] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *Proceedings of Learning @ Scale (L@S)*. 41–50. DOI: https://doi.org/10.1145/3051457.3051469
[35] Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering* 29, 4 (2003), 360–384. DOI: https://doi.org/10.1109/TSE.2003.1191799