Student Refactoring Behaviour in a Programming Tutor

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

Producing high-quality code is essential for professionals working on maintainable software. However, awareness of code quality is also important for novices. In addition to writing programs meeting functional requirements, teachers would like to see their students write understandable, concise and efficient code. Unfortunately, time to address these qualitative aspects is limited. We have developed a tutoring system for programming that teaches students to refactor functionally correct code, focussing on the method-level. The tutoring system provides automated feedback and layered hints. This paper describes the results of a study of 133 students working with the tutoring system. We analyse log data to see how they approach the exercises, and how they use the hints and feedback to refactor code. In addition, we analyse the results of a student survey. We found that students with some background in programming were generally able to identify issue in code and solve them (on average 92%), that they used hints at various levels, and we noticed occasional learning in recurring issues. They struggled most with simplifying complex control flow. Students generally valued the topic of code quality and working with the tutor. Finally, we derive improvements for the tutoring system to strengthen students' comprehension of refactoring.

ACM Reference Format:
Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student Refactoring Behaviour in a Programming Tutor. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20), November 19–22, 2020, Koli, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3428029.3428043

1 INTRODUCTION

Learning to program has been a major area of research for many decades [22, 27]. Researchers have studied the mistakes that students make [5], the misconceptions they have [25], and how we could help them solve their mistakes and correct misconceptions using various teaching approaches [33]. While a major goal is to write code that is functionally correct, it is also important that the code is understandable, concise and efficient. These aspects have been receiving less attention, although we have noticed an increase in studying the non-functional aspects of code.

Studies have found numerous qualitative issues in student code [9, 10, 17]. If we assess students solely based on functional behaviour through test cases, they might not see the importance of writing high-quality code. However, at a certain point, students will have to write larger programs together with other students, for which they need to use and adjust existing code. The need to analyse and refactor low-quality code will then become apparent. It is therefore vital to introduce students to the concept of code quality, raise awareness of its importance, and introduce them to code improvement in an accessible way.

Unfortunately, it is well-known that universities and other learning institutes struggle to give each student the personal attention and feedback they need, due to growth in enrolment figures and limited staffing means [6]. Tools can give students some complementary support. We have designed a tutoring system for programming that teaches students to refactor functionally correct programs. The system focuses on method-level refactorings, such as rewriting a complex expression, removing unneeded code, and replacing a language construct by a more suitable alternative. The functionality of the tutoring system is based on input from teachers and how they would want students to rewrite their code, which we investigated in an earlier study [18]. The system offers refactoring exercises: it is the student's task to rewrite functionally correct code. The student can check the program against test cases to ensure it still works correctly, and ask for hints with increasing detail if he or she does not know how to proceed.

This paper describes the results of a study of 133 students working with the tutoring system. The students are more experienced novices who have already taken programming courses before. We analyse log data to find out how they use the system, and if they are able to identify issues and rewrite the programs. We also describe the results of a student survey on the tutor. We discuss the findings on how students refactor and how the system can help them, as well as future improvements and implications for its use.

The contributions of this work are: (1) a first exploration of how students refactor code; (2) an analysis of the issues they struggle with and which hints they need to deal with those issues; (3) insight into how students value code quality.

This paper is organised as follows: Section 2 discusses related work. Section 3 describes the tutoring system. Section 4 describes the method. The results are shown in Section 5, and discussed in Section 6. Section 7 concludes and describes future work.

2 BACKGROUND AND RELATED WORK

2.1 Code quality in education

Our goal is to make students who already have some basic programming knowledge aware of the qualitative aspects of their programs, and teach them how to refactor their programs to make them easier to understand, more efficient, with the best use of language constructs. We define code refactoring as improving code step by step while preserving its functionality [13]. We focus on single methods and how to improve their directly observable properties such as control flow, expressions, and choice of language constructs. Layout, naming and commenting are outside our scope.

Maintaining high-quality code is a major topic in the field of software engineering. There is evidence of the persistence of code smells in large software systems (e.g. [29]). For programmers to produce good code it is vital that they learn to be aware of flaws and refactor code as soon as possible. We should therefore incorporate this into our Computer Science curricula, which has not always been done [4]. Kirk et al. [21] studied learning outcomes of 141 introductory programming courses, and found that for 71% of these courses code quality is not part of the learning outcomes. For the courses that do mention code quality in learning outcomes, it is unclear what exactly is being taught.

Several studies investigated non-functional problems in student code (e.g. [9, 10, 17]), from which we learn they are evident and often remain unfixed. Studies show mixed results regarding whether students address issues, which is apparently more the case when they are being assessed on it. In cases where they are not, issues remain present (e.g. [17]).

2.2 Tutoring systems for programming

There has been a wealth of studies describing digital tools and environments that help students with learning programming [8, 20]. These tools enable students to learn whenever and wherever they want, and alleviate teacher workload. Many of these systems support the student with (automated) feedback. Feedback is an essential aspect in teaching [14, 28], having the potential to exert great influence on learning, assuming it is delivered in an appropriate manner. Feedback can be *summative* (focused on the outcome) or *formative*, the latter defined by Shute as 'information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning' [28]. While assessment tools are mainly focussed on grading programs and giving summative feedback on final submissions, (intelligent) programming tutors help students during the stepwise process of solving exercises.

Intelligent Tutoring Systems (ITSs) have been studied extensively for various domains [31]. VanLehn found that ITSs were nearly as effective as human tutors [32]. ITSs have an *inner loop*, giving stepwise hints and feedback, and updating the student model; some ITSs also have an *outer loop*, selecting a suitable next task. Focussing on the inner loop, several aspects are important for offering feedback and hints [31]. A hint should preferably be given when a student really needs it, but it can be tricky to predict and control this. The suggested step should be analogous to what the teacher would advise, but should also support the student if he or she has already embarked on a certain solution path. The manifestation of hints is often gradual: a general hint, followed by a more descriptive hint, and finally a *bottom-out hint*, which is the actual step to be taken. Feedback usually consists of simply indicating correctness or incorrectness, and error-specific messages. For the domain of programming, several ITSs have been developed that offer features supporting both the inner loop and the outer loop [8]. These Intelligent Programming Tutors often teach a specific aspect (such as recursion), or support building small programs.

2.3 Automated feedback on code quality

In a paper that describes the details of the design of our tutoring system, we argue why professional code quality tools are unsuitable for novice programmers [19]. One type of such a tool is the static analyser that reports on violated issues in source code, which can be run inside an IDE or standalone. Examples are PMD, Checkstyle, SonarQube, and linters. Problematic for novices are the technical terminology, and the possibly very long lists of reported issues that are not always relevant in the context of novice programs. Other types are refactoring tools and other code transformation tools, often integrated in IDEs. These tools execute refactorings in one step, giving little insight into the inner workings. Some IDEs offer numerous code edits, often without a clear goal. Our tutoring system aims to overcome these problems by offering a student-friendly introduction to code refactoring, using understandable language, and layered feedback for a selection of relevant issues.

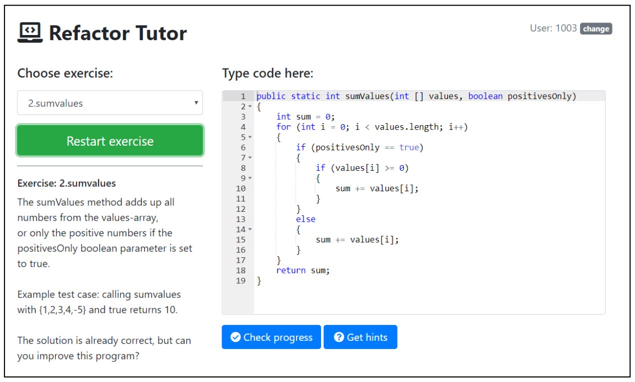
Professional tools have been used in the context of education though, and are surely relevant for the more experienced programmer [11, 24]. Jansen et al. [15] have used the Better Code Hub (BCH) tool in education. This tool checks a codebase in GitHub against ten software engineering guidelines, such as 'write short units of code' and 'write code once', finding some evidence of increased code quality, although the opinions of students about the tool varied.

There are also systems analysing code quality specifically aimed at students. FrenchPress [3] is an Eclipse plugin that checks Java code for seven issues related to misuse of fields, the public modifier, booleans, and loop control variables, and presents student-friendly messages. It was used in a trial by around 45 students for four exercises, with the result that between 36% and 64% self-reported that they would change their code according to the feedback received.

The Style++ tool provides students with a report on style issues such as commenting, naming and code size [1]. The tool has been used and evaluated by a great number of students, but how students use the tool has not been studied. The authors have seen an increase in the quality of programs, but that could also be partly attributed to the fact that submitted programs had to be approved by the tool.

WebTA [30] is a programming environment in which students receive feedback continuously, including stylistic issues. Examples most related to our system are useless language constructs, and declaring variables or resources inside a loop. We cannot derive from the paper how feedback is presented to students, and the effect of using the tool has not been measured yet. The Java Critiquer [26] points students to quality problems in their code. We could not find a study on student use.

AutoStyle [7] gives stepwise, data-driven feedback on how to improve the style of correct programs, consisting of teacher-written hints on clustered programs and automatic hints on features that could be added or removed. An experiment with students using the system has shown improvements, especially in recognising good coding style, but students also still struggled with improving code [35]. AutoStyle is different from our tool because it relies on historical student data, which has the disadvantage that this



Exercise: 2.sumvalues

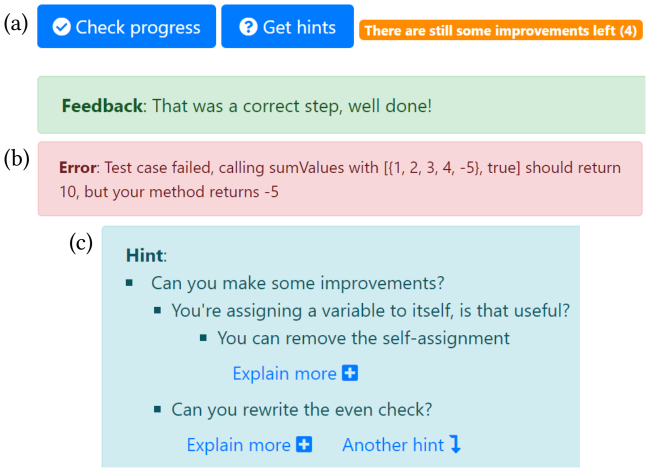
The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with {1,2,3,4,-5} and true returns 10.

The solution is already correct, but can you improve this program?

```java
public static 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;
}
```

Figure 1: Web application for the tutoring system.



Feedback: That was a correct step, well done!

Error: Test case failed, calling sumValues with [{1, 2, 3, 4, -5}, true] should return 10, but your method returns -5

Hint:
- Can you make some improvements?
  - You're assigning a variable to itself, is that useful?
    - You can remove the self-assignment
  - Can you rewrite the even check?

Figure 2: (a) Feedback acknowledging a correct step and an indicator of the number of improvements left, shown after clicking the CHECK PROGRESS button. (b) Error message for a failed test case. (c) Partly collapsed hint tree.

data is not always available, and requires teachers to write hints beforehand. Moreover, different from the AutoStyle studies, we have performed a quantitative study of log data on how students approach code improvements at the method level. We are not aware of any other such study.

3 THE REFACTOR TUTOR

This section describes the tutoring system for refactoring. The target audience are students (typically CS majors) who already know the basics of programming (control structures, loops, arrays, methods, etc.). The system offers exercises in which a problem specification and a functionally correct, but problematic program is given. The student's task is to improve (refactor) the program to make it more concise, efficient, and understandable.

Fig. 1 shows the web interface of the system. To implement the code editor we used the open source Ace editor[1], which supports syntax highlighting, automatic indentation, highlighting matching parentheses, code folding, and more. The student has two ways to ask for feedback during programming: CHECK PROGRESS and GET HINTS. The CHECK PROGRESS button checks the current status of the program, resulting in one of the following diagnoses:

- **Expected**. The system recognises a step, see Fig. 2a.
- **Correct**. The submitted program is functionally correct, but the system does not know what has been done.
- **Similar**. Nothing has changed, or the student went back to the previous state after doing an incorrect edit.
- **Buggy**. The system recognises a known incorrect step.
- **Failed test case**. The functionality of the program has changed. The first failed test case is shown, see Fig. 2b.
- **Compiler error**. The system detects an unsupported language construct or a syntax error, showing the number of errors and the first Oracle Java compiler error message.

For the first three diagnoses (dealing with a correct program) the system also shows how many improvements are left (see Fig. 2a).

The GET HINTS button generates hints for the current program state, after checking that the program is still correct by executing a progress check in the background. If it is not correct, the diagnosis is shown, otherwise hints are generated. The system presents hints in a tree structure, of which the first option is shown by default. The student can expand a hint (by clicking EXPLAIN MORE) to get a more detailed hint, or click on ANOTHER HINT to get a different hint. An example of a partly collapsed hint tree is shown in Fig. 2c.

The hints the system gives are based on rules, derived from teacher suggestions for a set of imperfect student programs, collected in our earlier study that investigated how teachers would give feedback on improving code [18]. Other rules are based on other studies [9, 36], a subset of rules from professional static analysis tools considered suitable for novices, and equality rules from arithmetic and logic. The system also contains some 'buggy rules' that describe common mistakes. The system, its motivation, design, and example sessions are described in more detail elsewhere [19].

The system can be accessed online[2], and consists of a web-based interface and a backend that processes JSON requests and replies with JSON responses. All requests and responses are logged in a database: retrieving the exercises, loading an exercise, checking progress, asking for hints, and expanding a hint. The current state of the code is attached to the requests. Two teachers and a TA tested the system before the experiment. It offers six refactoring exercises with varying difficulty. The programs contain between two and four quality issues at the start, see Table 1. These issues correspond to the rules described earlier. Some issues become apparent after dealing with initial issues. Certain issues reappear in a later exercise, sometimes in a slightly different way. The last exercise (exercise 6) has no starting code, only a description of its functionality and a set of test cases. Feedback and hints are available for all exercises, and are generated dynamically for the student code. Listings 1–2 show the start code and description for the first two exercises. Exercise 2 is taken from another study [23], exercise 1 and 6 are from the Codingbat website[3], and exercise 3, 4 and 5 are our own.

[1]ace.c9.io/

[2]www.hkeuning.nl/rpt

[3]www.codingbat.com/java

Table 1: The issues that appear (●) or may appear later (○) in the exercises.

	Ex1	Ex2	Ex3	Ex4	Ex5
Expressions					
Simplify boolean expression	-	●	●	-	-
Use compound operator	●	-	●	●	-
Optimise calculation	-	●	-	-	-
Remove self-assign	●	-	●	-	-
Improve odd/even check	●	-	-	-	-
Branching					
Remove duplication	-	●	-	-	-
Remove useless else if	-	-	●	-	●
Remove empty statement	○	-	○	-	-
Extract from if else	-	-	-	●	-
Loops					
For to foreach	●	●	-	-	-
For to while	-	-	-	-	●
Exit loop early	-	-	○	-	○
Replace loop by calculation	-	-	-	●	-

4 METHOD

Our research questions are:

RQ1 How do students solve refactoring exercises? Which steps do they take, and which mistakes do they make?

RQ2 When do they ask for a hint? For which issues do they need hints? How do they respond to a hint?

RQ3 What do students think about working with a refactoring tool?

We conducted an experiment at Windesheim University of Applied Sciences in the Netherlands in the week of 14 October 2019. This week was the final week of the term for (mostly) second-year IT-students specialising in Software Engineering, who were all doing a C# programming course. A minority of the students are specialising in other fields, such as embedded systems or business IT. All students had followed at least two previous programming courses: web programming in PHP, and object-oriented programming in Java. The course assumes the programming knowledge from these courses, in which all basic language constructs (variables, branching, loops, methods, object-oriented concepts, etc.) are discussed. The C# course transfers to the C# language, first discussing the C# equivalents for known language constructs, and then handling more advanced topics such as delegates and events, generics, LINQ, functional programming constructs and unit testing.

The course was taught by four different lecturers in seven groups of approximately 25 students. One of the researchers is a colleague of these lecturers. The experiment lasted one hour, in which the lecturer and one researcher were present. All students worked with the system. We did not offer a pre- and post-test for this study, because we want to focus on how students use the tool, how they respond to feedback and hints, and how they edit the code.

The experiment consisted of three parts: the first 15 minutes were used to introduce the topic of code quality, to demonstrate the tool, and to explain the experiment. The students were asked to fill in a form to provide consent for using their data, and to give some general information such as age, gender and previous knowledge. All students were given a unique ID to login to the system. If they did not give consent to use their data, they could log in anonymously and still do the exercises. In the next 30 minutes the students individually worked on six exercises on their own laptop. The students received an information sheet with some notes on how the system works, and the Java syntax for certain language constructs supported by the system. The system's interface and its feedback is in English. This is not the students' native language, but the students use other English study material as well. The researcher helped with questions on how the tool works, but not with questions on how to solve the exercises. The students were also asked not to consult other students. The final 15 minutes were spent to fill in a short survey with questions about their experience with the system. The survey had three Likert-scale questions and three open questions.

After the experiment the log database was cleaned by removing records from anonymous IDs and IDs that did not give consent. We also removed activity from outside the 30 minutes of the experiment, extended with an overrun of 10 minutes. We checked the dataset for abnormal activity and removed log sequences with more than 50 identical actions per minute. We suspect some of those sequences are not from the user interface, but were fired by a script.

Because the number of program states in this study is large, we performed an automated analysis. To check whether our system correctly identifies programs as 'ready', we manually inspected a random subset of 10% of the ready end states for each exercise.

5 RESULTS

In total 143 students (of around 200 enrolled, and some who had to retake the course) attended the experiment, of which 135 students gave consent to use their data, and 8 did not. For 1 of the 135 students that gave consent we could not find log data, and all records from 1 student were removed due to abnormal behaviour (an excessive number of requests), resulting in 133 students for the analysis. The students produced 12,254 log entries. The main events are summarised in Table 2.

The students were between 17 and 31 years old (average 20.5, median 20, 1 student did not provide age). A total of 86% identified as male, 8% as female, and 5% did not provide their gender. All but 1 student attended the web programming course and 130 (98%) passed. All but 4 students attended the Java programming course and 122 (92%) passed. Of all students, 10% reported they had no programming experience besides school, 60% had a little, 13% had a lot, and 23% had experience from a previous education. A total of 122 students (92%) are in their second study year and have chosen a software engineering profile, 2 students (2%) are retaking the course, and 9 students (7%) are IT students with a different profile that are taking the course as part of an elective minor.

5.1 Solving exercises (RQ1)

Fig. 3 shows the number of students who started and completed the exercises. We only include attempts for an exercise with at least one action (such as check or hint request). We define 'ready' as the system not having any hints left, 'gave up' as performing at least one action, but not dealing with all issues and moving on to a new exercise, and 'time up' as working on the exercise when the experiment stopped. For 'gave up' we also calculated the number

Table 2: Summary of tutor events, between parentheses the number of unique students.

Exercise	Startups		Diagnoses		Generated hint trees		Hint expansions	
1 Even	228	(133)	1537	(133)	205	(97)	122	(53)
2 SumValues	222	(133)	2539	(133)	456	(112)	573	(99)
3 OddSum	167	(123)	1329	(118)	261	(86)	226	(59)
4 Score	120	(103)	775	(97)	121	(51)	88	(30)
5 Double	93	(76)	402	(69)	55	(27)	26	(10)
6 HaveThree	75	(60)	376	(44)	18	(11)	14	(6)
Total	*905*		*6958*		*1116*		*1049*	

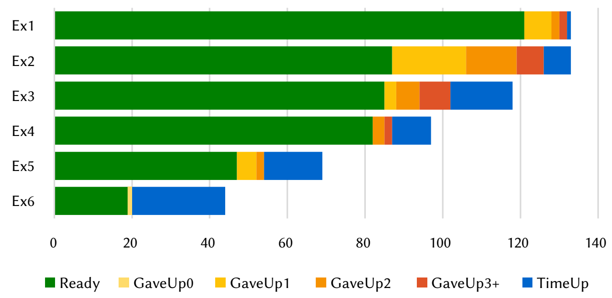


Figure 3: For each exercise the number of students that solved all issues (green), did not solve all issues and continued with another exercise (yellow/orange/red, depending on the number of open issues), and were working on an exercise when their time was up (blue).

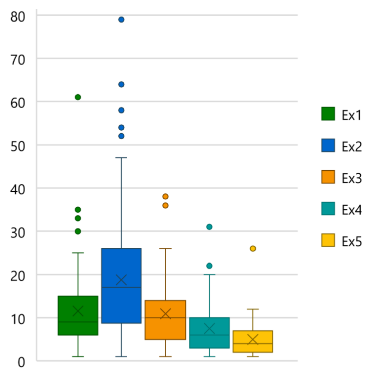


Figure 4: For each exercise a boxplot showing the distribution of the number of diagnoses per student.

of hints remaining for the last known and valid program state. The first two exercises were started by all students, with a gradual decline for the subsequent exercises. In total 52% of the students got to exercise 5, and exercise 6 was attempted by only 33%, although 45% started it. This is probably due to the fact that students had to write code from scratch in exercise 6. We exclude exercise 6 in the remainder of this paper, because of its different nature, and low number of students that attempted it. Exercise 2 and 3 have by comparison the most students that did not complete it.

Table 3: Time on task.

Exercise	Min	Max	Mean	Median
1 Even	1:33	17:55	7:06	6:44
2 SumValues	1:38	27:34	10:12	9:33
3 OddSum	1:54	16:15	6:41	6:23
4 Score	1:11	14:06	4:41	4:14
5 Double	1:05	13:45	3:49	3:20

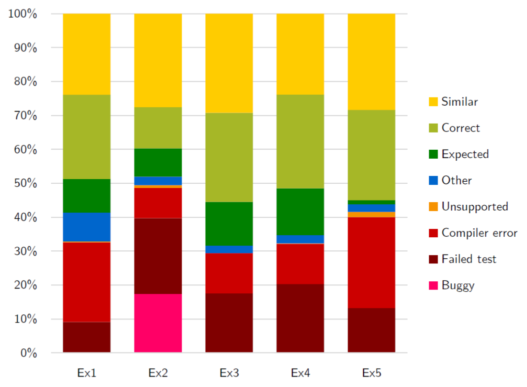


Figure 5: For each exercise the distribution of diagnoses.

Table 3 shows for each exercise how much time students spent working on them, excluding timeups. Students worked at least between 1 and 2 minutes on all exercises. They spent on average most time on exercise 2.

Next we zoom in on the diagnoses students received while working on the exercises. A diagnosis is calculated when a student clicks on CHECK PROGRESS, but also when a student asks for a hint, because hints are only generated once the current program state is functionally correct. Again, we only include sessions in which the student has performed at least one action. We exclude timed-out sessions because their diagnose count would not be representative for a full session. Fig. 4 shows how many diagnoses a student receives, which varies per exercise with a median between 4 and 17. We manually inspected some of the outlier sessions, in which we often noticed a large number of identical error diagnoses given in a short time frame, giving the impression that the student kept clicking the button again and again.

Fig. 5 shows the distribution of the various diagnoses. Table 4 shows the diagnoses with a functionally correct result, and Table 5 the diagnoses with a problematic result. For exercises 1 to 4 we see a fair amount of expected (a single recognised step) diagnoses in relation to correct diagnoses (multiple or unknown steps resulting into a correct program). This is much less the case in exercise 5.

Failed test was a major category (22%) of the problematic diagnoses for exercise 2. Failed test can be an actual test case failing, the inability to execute a test because a student changed the method header, or some other runtime error such as a suspected infinite loop. Compiler errors are also pervasive. Note that one compiler error instance might consist of multiple compiler errors. The number of compiler errors is the largest in the first exercise, which could be attributed to getting used to the Java syntax. We also noticed students using language constructs unsupported by our tutoring

Table 4: For each exercise the total number of diagnoses for functionally correct solutions, and between parentheses the number of unique students receiving that diagnosis.

Exercise	Expected		Correct		Similar	
1 Even	152	(98)	382	(132)	367	(115)
2 SumValues	213	(103)	307	(120)	700	(116)
3 OddSum	172	(93)	349	(113)	388	(98)
4 Score	107	(61)	214	(92)	185	(62)
5 Double	5	(3)	107	(63)	114	(44)
Total	*6580*		*1359*		*1754*	

Table 5: For each exercise the total number of problematic diagnoses.

Exercise	Failed Test		Compiler error		Unsup-ported		Other	
1 Even	140	(45)	361	(101)	4	(3)	131	(46)
2 SumValues	567	(108)	226	(82)	22	(19)	63	(28)
3 OddSum	233	(68)	158	(71)	0	(0)	29	(16)
4 Score	157	(46)	92	(48)	1	(1)	19	(9)
5 Double	53	(27)	108	(44)	6	(3)	9	(5)
Total	*1150*		*945*		*33*		*251*	

system, such as the `?:` ternary operator (shorthand for an if), other unusual operators, and calling library methods.

The 'other' category are diagnoses that were less clear, and contained some internal errors caused by bugs. Students using constructs such as `return sum+=1` and `if(x=1)` with assignments in (boolean) expressions received messages that should have been clearer: the system dealt with these constructs insufficiently. We also noticed that students often (mostly in the first exercise) mixed a foreach with standard array indexing, causing an error message not suitable for the actual problem. We explain the 'buggy' category for exercise 2, accounting for 17% of the diagnoses, in Section 5.2.2.

5.2 Hint seeking (RQ1 and RQ2)

Fig. 6 shows per exercise how many hints students have seen, including the top-level hint of the initial hint tree, and hint expansions and alternatives. We exclude timed-out sessions and sessions with no activity. The most hints were seen for exercise 2, with a median of 7.5. The medians for the other exercises are at most 2, but there are quite some outliers with students viewing many more hints. The last three exercises show a decreasing number of hints.

Next, we focus on the individual exercises to investigate which hints were shown and if students were able to solve issues with or without help. We exclude students whose time was up for a particular exercise. Tables 6 and 7 show the main hints for exercises 1 and 2, and are discussed in detail in the next subsections. Recurring issues (sometimes in a slightly different manifestation) are indicated with a ↻. Issues that might come up later during the exercise are marked with a ★. For these later issues we only look at students who received a hint for it. Some other issues that came up incidentally because of undesirable student edits (for which hints are generated dynamically) are omitted from the tables and analysis.

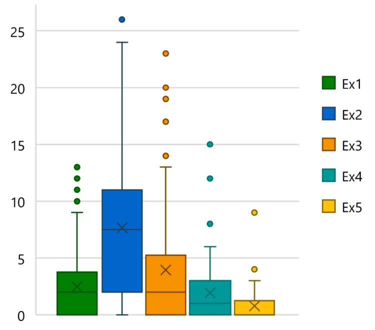


Figure 6: For each exercise a boxplot showing the distribution of the number of hints seen per student.

5.2.1 *Exercise 1.*

Table 6 shows the main hints associated with exercise 1, organised in the tree structure in which the hints were shown to the students. All hints in this table are known issues and form the complete hint tree generated for the start program. The table shows the number of students that have seen the hint, and how many students have solved the issue. The issue for which most hints were generated is replacing the for-loop by a foreach-loop, which was seen by 61% of the students. Almost a third of those students expanded that hint to see the code example. Some students did not make this change to the code in the end. A total of 43% of the students saw the hint on rewriting the even check, for which more than half expanded to see more detail, and almost a third of those saw the code example. A small number of students did not solve the issue. Most students removed a self-assignment statement and used a compound operator without help.

Manual inspection of 10% of the ready programs confirmed that all issues were dealt with, although one student used `+=1` instead of `++` and one program contained an unnecessary declaration, something the system cannot detect yet. The most common failed test case was often caused by replacing the statement `count=count` by `return count`, which is a curious misconception. We also saw some students using `values[i]%2==2`, which can never be true.

5.2.2 *Exercise 2.*

Table 7 shows the main hints for exercise 2, with which students struggled more. Most of the main hints were seen by more than half of the students, except replacing the for-loop by a foreach-loop. This was a recurring issue that, compared to the previous exercise, twice as many students changed without seeing a hint. In particular, students struggled a lot with the duplicated addition. More than two thirds of the students viewed some hint on this topic, and the majority of those students clicked all the way through to the code example. Those students were most likely (88%) to deal with the issue, but in the end only 75% of all students did.

The system contains a 'buggy rule' related to merging the two conditions for adding the array value (lines 5–12), which detects the common mistake of incorrectly combining the conditions, resulting in the code below:

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

Listing 1: Start code for exercise 1.

```
1  int countEven(int [] values) {
2    int count;
3    count = 0;
4    for (int i=0; i < values.length; i++)
5    {
6        if (values[i] % 2 != 1) {
7          count = count + 1;
8        }
9        else {
10         count = count;
11       }
12   }
13   return count;
14 }
```

Description: The countEven method returns the number of even integers in the values-array.

Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers.

The solution is already correct, but can you improve this program?

Listing 2: Start code for exercise 2.

```
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 }
```

Description: The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with {1,2,3,4,-5} and true returns 10.

Table 6: Hints seen and solved for exercise 1 (n=132). The 'solved by' column calculates the percentage based on the 'seen by' column.

Type of the most detailed hint seen	Seen by		Solved by		Total solved
Replace for by foreach-loop (line 4-12)					
No hint	52	(39%)	49	(94%)	
Top-level hint	54	(41%)	48	(89%)	121 (92%)
→ Code example	26	(20%)	24	(92%)	
Rewrite the even check using `==0` (line 6)					
No hint	75	(57%)	73	(97%)	
Top-level hint	19	(14%)	19	(100%)	128 (97%)
→ Detailed hint	26	(20%)	25	(96%)	
→ Code example	11	(8%)	10	(91%)	
Remove useless else with self-assign (line 9-11)					
No hint	119	(90%)	118	(99%)	
Top-level hint	6	(5%)	5	(83%)	130 (98%)
→ Detailed hint	4	(3%)	4	(100%)	
→ Code example	3	(2%)	3	(100%)	
Use the compound ++ operator (line 7)					
No hint	124	(94%)	123	(99%)	
Top-level hint	4	(3%)	4	(100%)	130 (98%)
→ Detailed hint	2	(2%)	2	(100%)	
→ Code example	2	(2%)	1	(50%)	

Table 7: Hints seen and solved for exercise 2 (n=126). * We do not calculate total solved for incidental issues, because not all students had to deal with them.

Type of the most detailed hint seen	Seen by		Solved by		Total solved
Remove duplication by simplifying ifs (line 5-12)					
No hint	35	(28%)	26	(74%)	
Top-level hint	10	(8%)	5	(50%)	94 (75%)
→ Detailed hint	12	(10%)	2	(17%)	
→ Code example	69	(55%)	61	(88%)	
Boolean expression issues					
No hint	43	(34%)	40	(93%)	
Top-level hint	14	(11%)	13	(93%)	
→ Detailed hint for remove == true (line 5)	11	(9%)	11	(100%)	123 (98%)
→ Code example	0	(0%)	–		
→ Detailed hint for complex expression ★	5	(4%)	4	(80%)	n.a.*
→ Code example	54	(43%)	51	(94%)	
Use > to avoid useless calculations (line 6)					
No hint	47	(37%)	43	(91%)	
Top-level hint	23	(18%)	18	(78%)	114 (90%)
→ Detailed hint	18	(14%)	17	(94%)	
→ Code example	38	(30%)	36	(95%)	
Replace for by foreach-loop (line 4-13) ↻					
No hint	113	(90%)	99	(79%)	
Top-level hint	5	(4%)	4	(80%)	109 (87%)
→ code example	8	(6%)	6	(75%)	

Our system confused students by reporting the name of this issue ('buggycollapseif') instead of an informative message. Excluding timed-out sessions, we counted 415 diagnoses of this issue for 80 students, who saw the message between 1 and 40 times with a median of 3. 49 (61%) students solved the issue, which is lower than the 75% of all students. Showing an explanatory message probably helps understanding what went wrong, and increases the number of students solving the issue.

Many students also saw some hint on improving a boolean expression, such as removing `==true`, which most students did without a hint. Another issue in this category that came up during refactoring, was simplifying a complex expression, usually appearing after merging the two conditions for adding the array value. Students probably used the code example from the duplication hint, which first showed the disjunction of the two cases. Most students needed the code example to see how the expression could be simplified, which involved applying logic rules. Of all students, 63% saw the hint on using > instead of >=, often also expanding to the code example. Students that only saw the top-level hint were least likely to address the issue. In the end 90% changed the operator.

Manual inspection of 10% of the 'ready' programs revealed that two students chose an alternative solution by always adding a value to the sum, and either setting that value to 0 in certain cases, or later subtracting the value if it should not have been added. We have seen this type of solution in teacher solutions as well, and it would be a good candidate to discuss its pros and cons. In two cases the >= was still present, but should have been detected, and one of those cases still contained the duplication in a different construct. The other 5 programs contained no issues.

5.2.3 Exercise 3 to 5. We only discuss notable observations from exercises 3 to 5, which were attempted by fewer students. The extensive analysis and descriptions for these exercises can be found in a PhD thesis [16].

For exercise 3, the issue not dealt with by the largest number of students (8) was removing a useless check in the else-if part of an if-else. All but one of the students that saw a hint for that issue fixed it, and for half of those students the top-level hint was sufficient.

The code for exercise 3 contained multiple recurring issues, such as a useless else with a self-assignment similar to the code for exercise 1. However, a slightly smaller percentage (95% versus 98%) dealt with it. A possible reason for this might be that the code for exercise 3 was much more complex, distracting from these useless lines of code. However, we also observe that the top-level hint was more often helpful compared to exercise 1. The `==false` is also changed less often than the `==true` from exercise 2, but that might be because this change is slightly more difficult. The compound operator += hint was seen by 50% of the students, which could also be due to the fact that this hint appeared early in the ordering of hints. Students asked for various levels of detail, and managed to use the operator in all but 2 cases. We expect that students are less familiar with this operator than the more common ++ operator.

Exercise 3 contains a loop that can terminate once -1 is seen in an array, on which hints are given once all clutter has been removed. Of all students, 38% have seen a hint for this. The system offers two alternative solutions, of which the first (adding the stop-variable to the condition of the loop) has been seen by more students than the second (directly checking if the current array value contains -1 in the loop condition). The majority of students seeing these hints have also dealt with the issue.

Manual inspection of 10% of the 'ready' programs showed that all issues were dealt with, except some issues with exiting from the loop once -1 was seen. Some students used a return or break: a good alternative to the system's suggestions. In a few of those cases the stop variable was maintained, which is unnecessary. One student used a foreach while maintaining a counter for skipping even indices: we consider converting this into a buggy rule.

The start code for exercise 4 contains an unnecessary for-loop that can be replaced by a simple calculation. Almost half of the students saw the top-level hint for this exercise, and quite a few also requested the code example. All of these students managed to replace the loop. The code also contained an if-else statement with the same code at the end of both the if-part and the else-part, which can be moved after the if-else. Around a quarter of the students viewed the hint, of which the majority needed more detail. Only one student did not fix it despite the hints. The recurring compound `-=` operator was now replaced more often than in the previous exercise, and a much smaller percentage needed a hint for this issue.

The start code for exercise 5 contains a recurring issue: the useless check in the else-if that also appeared in exercise 3. Here a larger percentage of students solved it, and the same percentage viewed a hint, but only 1 student requested more detail. Earlier exercises suggested to use a foreach instead of a regular for-loop, whereas this exercise suggests using a while because of the unknown number of iterations and the more complex stop condition. In the end, 87% changed the loop.

5.3 Student evaluation (RQ3)

All 133 students filled in a survey after the tutoring session. They answered 'what do you think about paying attention to code quality' on a 5-point Likert scale ranging from 'not important at all' to 'very important'. 24% answered very important, 67% important and 9% neutral. 3% of 131 students found the exercises very easy, 14% easy, 68% neutral, 15% difficult, and 0% very difficult. 13% found the hints very useful, 64% useful, 13% neutral, 3% not useful, and 0% not useful at all. Ten students answered that they did not use the hints.

The remaining subsections discuss the responses to the questions 'what did you like about the system' and 'what would you want to see different in the system' in four different categories.

5.3.1 Learning about code quality. Multiple students said something like 'fun', 'challenge' and 'good initiative'. Four students commented on the topic of code quality and its importance. One student wrote: 'Code quality is one of the most important things in a team. It should be stimulated more and the tool helps with that'. More than 40 students said something they liked about what they learned from working with the system, such as 'That you can see that the code looks simpler when you changed something and it is correct'.

Several students used the phrase 'making you think', such as 'It lets you think about solutions that you normally wouldn't think of'. Three students mentioned as an improvement that it would be useful to better explain what the benefit is of a certain suggestion.

5.3.2 Hints and feedback. About 48 students positively mentioned the hints and feedback the system gives. Most comments were about the usefulness and clarity of the hints. Six students appreciate that hints can be gradually revealed and do not give away the solution straight away: 'Often, the first was sufficient' and 'Hints only point in the right direction, so you'll come up with improvements yourself, unless you really can't'. Multiple students like the fact that they can track progress by seeing how many improvements can be made.

There are some issues with the feedback, which seems to be the case for a few specific instances, such as the 'buggy if collapse' (discussed in 5.2.2). Other students said the feedback could have been clearer, which was mentioned more often for error messages than for hints. Some students suggested to improve the feedback, for example by clearly indicating the location of errors. Students also pointed out some unexpected behaviour ('Errors where there should not be an error'). We also suspect some students were confused by Java compiler messages, a known and persistent problem [2].

5.3.3 User interface.

5.3.3 User interface. A total of 28 students mentioned the user interface in a positive way, mentioning its simplicity, usability and speed, or simply that it worked well. 22 students were unhappy that the current state for an exercise is not saved, so they lose their progress if they go to another exercise or accidentally leave the page. 9 students expressed their satisfaction with the code editor and its syntax highlighting, code formatting and useful shortcuts. The responses also contain suggestions for improving the editor (auto-completion, underlining syntax errors similar to what IDEs do), and multiple suggestions about improving the interface in general (smoother transition between exercises, undo/redo functions).

5.3.4 Exercises. 11 students particularly mention the exercises in a positive way: clear and not too big. 9 students would like to see some changes: 3 found some exercises unclear and the others all had their own requests, such as more (variety in) exercises and difficulty levels. 7 students would like to see C# support (or other languages), which they were mostly using at that time. 4 students requested support for features such as functions and ternary operators.

6 DISCUSSION

This section revisits the research questions and discusses our findings (6.1), reflects on the implications for teaching code refactoring (6.2), and discusses threats to validity (6.3).

6.1 Student refactoring behaviour

How do students solve refactoring exercises? Which steps do they take, and which mistakes do they make? (RQ1). The students managed quite well to solve refactoring exercises. However, because exercise 2 showed signs of struggle, fewer students completed the exercises after the second. On average, 92% of the quality issues that were present at the start were dealt with. Students regularly used the Check diagnosis function, which we view positively because a central aspect of refactoring is to maintain functional behaviour. The proportion of expected diagnoses shows that students also regularly took small steps. Compiler errors and failed tests are still pervasive, even though the students started with functionally correct code. We want to decrease the number of failed tests by adding more buggy rules to provide specific feedback on incorrect steps. For the first few exercises, the majority of the students saw at least one hint. In the subsequent exercises, fewer students saw hints.

When do they ask for a hint? For which issues do they need hints? How do they respond to a hint? (RQ2). The majority of the students regularly requested hints. For all issues in the code that were known at the start, on average 34% received a hint (mean 22%). It varied per issue whether a student expanded the hint to get more detailed information. Even though some researchers warn against providing progressive hints ending with the correct answer [28], we did not see negative effects of this. We told students to just ask for the hints when needed, and noticed from the data that they requested hints at various levels. For only a few complex issues the bottom-out hint was also the most requested. The fact that the participants were second-year students might have contributed to this behaviour.

Students had no problem with simple edits, such as using compound operators and removing self assignments, but did have trouble with complex control flow (nested if statements and breaking out of a loop) and composed expressions. Professional static analysis tools usually do not flag these more algorithmic issues, showing the need for educational tools that provide support for these issues.

For recurring issues we see progress in some cases. More students managed to solve them, or needed a hint with less detail. Not in all cases we noticed an effect, which could be due to a different context and the student's inability to transfer what they had done before.

What do students think about working with a refactoring tool? (RQ3). We noticed a positive attitude towards code quality, which might be expected from students in their second year that have chosen the software engineering profile. They found the level of the exercises appropriate for the amount of time and their skills. Overall, the students positively evaluated the system, and considered the hints to be useful. Some bugs and unclear (error) messages understandably confused them. These negative experiences might have caused resistance to the system. Making students aware and letting them think about code quality instead of just writing code to meet the requirements of an assignment seems to be a valuable effect of using the tool. However, from the feedback we also learn that discussing the improvements the tool suggests is important.

6.2 Teaching code refactoring

Students can solve refactoring exercises, and from their comments we derive it has made them think about a topic they think is important, but normally do not spend much time on. We think better results can be obtained by accompanying the tool with instructions on code quality, and discussing the suggestions of the tool afterwards. A few students mentioned they disagreed with certain 'improvements'. We might discuss the nature of the various refactorings, varying from stylistic, such as using the compound operator *=, to more algorithmic. We should warn that the system might introduce them to language constructs that they might not have encountered yet. For example, using foreach instead of a for-loop, if possible, has benefits and can protect the programmer from accidentally doing something unwanted. We want to incorporate this in the tool, and make it part of the programming instruction.

Research on using educational code quality tools with students is scarce. Wiese et al. [35] gave an experimental group feedback from their AutoStyle tool, and a control group only a quality metric (the ABC-metric [12]). Both groups took a pre-test and a post-test, and the authors described some case studies of particular student types. Because their students started out with their own code, and our study focusses more on how students worked in the tutor, we cannot directly compare results. However, both studies support the potential for using these kinds of systems, the call for more in-depth help for students, and more studies evaluating the results.

6.3 Threats to validity

The system contained some bugs, which confused some students and might have slowed down their progress. Because the students were learning a different programming language at the time of the experiment, they probably needed time to recall Java syntax.

The granularity of the log data is at snapshot-level, which is in between submission-level and key-stroke-level, as identified by Vihavainen et al. [34]. The snapshots were recorded when students did a check or hint request, implying that we do not have detailed information on all of their steps, and our analysis could have missed relevant edits that were undone before taking an action. Solving an issue is measured by calculating hints for subsequent program states, and determining the differences. Some issues might be solved in unexpected ways for which our system does not generate a hint, which could mean in some cases there was no actual improvement.

Another threat to validity is that fewer students worked on the last few exercises, which are probably the 'better' students. This could cause an overestimation on how well students performed on these exercises, and would explain the fact that fewer students received hints for these exercises. The novelty of a new tutoring system could also have an effect on the enthusiasm of the students.

7 CONCLUSION AND FUTURE WORK

This paper describes the results of an exploratory study of log data from students working on refactoring exercises, and the students' personal experiences thereof. The students worked on six exercises in which they had to improve imperfect example solutions. All participants had at least a basic background in programming, and were mostly able to do the exercises, with regular checks to verify correctness. However, they struggled with complex control flow. The students regularly requested hints, at various levels of detail. After seeing a hint about a particular issue, most students solved the same issue without a hint when they encountered it again. Overall, students valued the topic of code quality and working with the system, but also proposed valuable enhancements.

These results contribute to focussing the attention of teachers and tool designers, by incorporating the discussion of refactoring rules in education (in particular those related to complex expressions and control flow), paying attention to alternative language constructs, and providing feedback at various levels to meet the needs of individual students. We will continue to improve and validate the system with students to make them more aware of the quality of code and the importance of refactoring. We intend to extend the ruleset and add more buggy rules. We also want to refine feedback messages and hints with more explanations on why an edit is useful. Support for methods could present new opportunities to improve code. After incorporating improvements, we intend to conduct an experiment with a control group and experimental group, and a pre- and post-test to determine learning gains. We also want to study the effect of using different types of feedback.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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.

[2] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *ITiCSE WG Reports*. 177–210.

[3] Hannah Blau and J. Eliot B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *ITiCSE*. 15–20.

[4] 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 *ITiCSE WG Reports*. 70–85.

[5] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *TOCE* 17, 2 (2017).

[6] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. 2003. How shall we assess this?. In *ITiCSE WG Reports*. 107–123.

[7] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-Driven Automatic Hint Generation for Coding Style. In *ITS*. 122–132.

[8] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *ACE*. 53–62.

[9] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding Semantic Style by Analysing Student Code. In *ACE*. 73–82.

[10] Stephen Edwards, Nischel Kandru, and Mukund Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *ICER*. 65–73.

[11] 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 *HICSS*. 7825–7834.

[12] Jerry Fitzpatrick. 1997. *Applying the ABC Metric to C, C++, and Java.* Technical Report. https://www.softwarerenovation.com/ABCMetric.pdf

[13] Martin Fowler. 1999. *Refactoring: improving the design of existing code.*

[14] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of Educational Research* 77, 1 (2007), 81–112.

[15] Julian Jansen, Ana Oprescu, and Magiel Bruntink. 2017. The Impact of Automated Code Quality Feedback in Programming Education. In *SATToSE*.

[16] Hieke Keuning. 2020. *Automated Feedback for Learning Code Refactoring.* Ph.D. Dissertation. http://www.hkeuning.nl/PhD%20Thesis%20Hieke%20Keuning.pdf

[17] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *ITiCSE*. 110–115.

[18] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How Teachers Would Help Students to Improve Their Code. In *ITiCSE*. 119–125.

[19] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A Tutoring System to Learn Code Refactoring. (2021). To appear in SIGCSE.

[20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *TOCE* 19, 1 (2018).

[21] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On Assuring Learning About Code Quality. In *ACE*. 86–94.

[22] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *ITiCSE WG Reports*. 55–106.

[23] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the Differences Between Correct Student Solutions. In *ITiCSE*. 177–182.

[24] 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.

[25] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *TOCE* 18, 1 (2017).

[26] Lin Qiu and Christopher Riesbeck. 2008. An incremental model for developing educational critiquing systems: experiences with the Java Critiquer. *Journal of Interactive Learning Research* 19, 1 (2008), 119–145.

[27] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

[28] Valerie J. Shute. 2008. Focus on formative feedback. *Review of Educational Research* 78, 1 (2008), 153–189.

[29] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *ICSE*. 403–414.

[30] Leo C. Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. In *SIGCSE*. 738–744.

[31] Kurt VanLehn. 2006. The Behavior of Tutoring Systems. *J. of AIED* 16, 3 (2006), 227–265.

[32] Kurt VanLehn. 2011. The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. *Educ. Psych.* 46, 4 (2011), 197–221.

[33] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *ICER*. 19–26.

[34] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. 2014. Analysis of source code snapshot granularity levels. In *SIGITE*. 21–26.

[35] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In *L@S*. 41–50.

[36] Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Trans. on Software Engineering* 29, 4 (2003), 360–384.