Improving Debugging Skills in the Classroom – The Effects of Teaching a Systematic Debugging Process

Tilman Michaeli
Computing Education Research Group
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstraße 3, 91058 Erlangen, Germany
tilman.michaeli@fau.de

Ralf Romeike
Computing Education Research Group
Freie Universität Berlin
Königin-Luise-Str. 24-26, 14195 Berlin, Germany
ralf.romeike@fu-berlin.de

ABSTRACT

Debugging code is a central skill for students but also a considerable challenge when learning to program: helplessness and, in consequence, frustration when confronted with errors is a common phenomenon in the K12 classroom. Debugging is distinct from general programming abilities, therefore it should be taught explicitly. Despite this, debugging is an underrepresented topic in the classroom as well as in computer science education research, as only few studies, materials and concepts discuss the explicit teaching of debugging. Consequently, novices are often left on their own in developing debugging skills. This paper analyses the effectiveness of explicitly teaching a systematic debugging process, especially with regard to the students' self-efficacy and the resulting debugging performance. To this end, we developed an intervention, piloted it and then examined it in a pre-post-control-group-test-design: Both experimental and control groups were surveyed using a questionnaire and given debugging exercises as a pre-test. Afterward, the intervention was carried out in the experimental group, while the control group continued to work on debugging exercises. During the post-test, the students once more worked on debugging exercises and were surveyed. The results show a significant increase in both self-efficacy expectations and debugging performance in the experimental group in contrast to the control group. Therefore, our study provides empirical arguments for explicitly teaching debugging and simultaneously offers a hands-on approach for the classroom.

CCS CONCEPTS

• **Social and professional topics → K-12 education**;

KEYWORDS

debugging, teaching practice, CS education, intervention study

ACM Reference format:
Tilman Michaeli and Ralf Romeike. 2019. Improving Debugging Skills in the Classroom – The Effects of Teaching a Systematic Debugging Process. In *Proceedings of WiPSCE '19, Glasgow, Scotland, October 23-25, 2019,* 7 pages.
https://doi.org/xxx

1 INTRODUCTION

Programming requires a multitude of competencies and their teaching represents a central challenge in computer science education. Learners do not only have to understand programming concepts, but they also need to be able to independently find solutions when being confronted with errors – an essential skill in the context of programming: Systematically examining programs for bugs, finding and fixing them is a core competence of professional developers. They spend between 20 and 40 percent of their working time debugging [27]. However, dealing with errors is particularly difficult for programming novices and poses a major obstacle when learning to program [22]. As programming novices tend to make more errors, this often is a major source of frustration [26]. Effective debugging therefore facilitates the process of learning to program.

Moreover, debugging skills do not only play a major role in the programming domain: debugging is also ubiquitous in our everyday life and research findings indicate that the explicit teaching of debugging can result in a transfer of debugging skills to a non-programming domain [8]. Recently, debugging has gained an increasing amount of attention in the context of computational thinking [35]. Thus, debugging is prominently found in recent curricula that build upon computational thinking such as the British "Computing Curriculum" [7].

Despite the importance of teaching debugging, there are surprisingly few studies dealing with the explicit teaching of debugging skills. At the same time, teachers lack concepts and material for teaching debugging. Instead, they focus on content such as algorithmic concepts or language syntax when teaching programming [24]. Therefore, novices are often left alone with their errors and, consequently, are forced to learn debugging "the hard way".

The aim of this intervention study is to examine the influence of the explicit teaching of a systematic process to debugging. With the focus being on the influence of this intervention on students' *self-efficacy* and *debugging performance*. For this purpose, we developed and piloted an intervention and examined it in a pre-post-control-group-test-design.

This paper is structured as follows: In section 2, related research regarding a systematic debugging process and debugging in the classroom is discussed. In section 3, we outline the methodology of our intervention study and the pre-post-control-group-test-design. Afterward, we present our results regarding the effects on self-efficacy (RQ1) and debugging performance (RQ2). In section 5, the results are discussed and in section 6, we present our conclusions.

2 RELATED WORK

2.1 Debugging skills

Debugging describes the process of finding and fixing errors. Debugging skills differ from general programming skills, as [1] or [12] show. They found, that, while pronounced debugging skills usually indicate corresponding skills in programming, the reverse is not necessarily true: Good programmers are not always good debuggers. This raises the question: What makes a "good" debugger?

The use of **debugging strategies** – sometimes referred to as tactics – plays an important role in the debugging process [11]: strategies such as tracing the control flow using *print-f* debugging, commenting out code or slicing can provide information that helps to localize the error [31]. One of the main differences between experts and novices is the efficiency of applying debugging strategies [23]. Murphy et al. [25] give an overview of strategies commonly employed by novices.

Furthermore, the proficient usage of **tools** – most prominently, but not limited to the debugger – can support the debugging process in a similar way as strategies and is, therefore, also seen as an integral debugging skill.

Often, there is no need to apply advanced debugging strategies or tools in order to find and fix errors: With experience, and therefore the **application of heuristics and patterns**, typical errors and their possible causes are more easily found. To support this "learning from previous mistakes", many professional developers maintain a debugging "diary" to document their debugging experience [27].

The application of a **systematic debugging process** – sometimes referred to as strategy – i.e. the structured pursuit of a high-level plan, represents the fundamental skill upon which the aforementioned skills are built. For professional debuggers, a large series of process models can be found (such as [13, 14, 19, 31, 34, 36]). These models agree on the following aspects: After testing the program and becoming aware of errors, hypotheses are repeatedly formulated, verified in experiments and, if necessary, refined until the cause of the error is found. This procedure is based on the so-called *scientific method* [36], which is typically implicitly applied by professional developers [27].

2.2 Error Types

Depending on the type of the underlying error, the debugging process varies greatly. This is mostly caused by differences in information available to the programmer. For example, for errors found by the compiler, line number and error message are provided. Therefore, the step of locating the error is often redundant. In contrast, for logical errors, no information is given. Here, locating the error is the most difficult step [14].

There is a variety of error type categorizations (cf. [17, 18, 21, 33]). Typical classifications of errors distinguish between syntactic (mistakes in spelling or punctuation), semantic (mistakes regarding the meaning of the code) and logical errors (arising from a wrong approach or a misinterpreted specification), compile-time and run-time errors, or differentiate between construct-related (in the sense of programming language specific constructs) and non-construct-related errors.

Studies on error frequency for novices commonly focus either on specific error types (such as errors detectable by tools or the compiler), exclusively consider the final versions of programs (and therefore omitting interim versions), or feature only small sample sizes. For this reason, it is difficult to make quantitative statements based on literature alone. Altadmri and Brown [4] give an overview of bugs commonly encountered by novices in Java, based on the analysis of the BlueJ-Blackbox data set. Their study shows that overall semantic errors occur more frequently than syntactic ones, particularly once users show a certain degree of programming proficiency – although a syntax error was the most common error found. When they do occur, these syntactic errors are fixed very quickly, due to their low error severity (typically measured based on the time needed to fix an error). In a study with a limited data set, Hall et al. [16] showed that logical errors seem to be the most widespread type. Spohrer and Soloway [32] add to this that non-construct-related errors occur more frequently than construct-related ones, contrary to popular belief. Errors the compiler does not identify are considered much more difficult – and therefore time-consuming – to find and fix [4, 25]. Nevertheless, teachers report that compile-time errors also pose a major problem for novices, especially in the K12 classroom and therefore need to be addressed in teaching materials [24].

2.3 Teaching Debugging

Murphy et al. [25] as well as Kessler and Anderson [20] argue that debugging skills should be explicitly taught. Despite this, there are surprisingly few studies, both on university teaching and in K12, which deal with the explicit teaching of debugging.

2.3.1 *Debugging in Higher Education.*

Chmiel and Loui [9] used voluntary debugging tasks and debugging logs to promote students' debugging skills. It turned out that students who had completed the voluntary debugging tasks needed significantly less time to debug their own programs. However, this correlation was not reflected in the exam results, which, contrary to expectations, were only slightly better.

Katz and Anderson [19] investigated the effect of teaching different debugging processes, such as forward-reasoning and backward-reasoning, for debugging LISP programs. Different groups of students were first taught one of the approaches before they were free to choose their own procedure. This showed that students continued to use the procedure they had been taught.

Allwood and Björhag [3] investigated to what extent written debugging instructions could support the process. While the number of bugs did not differ between the experimental and control groups, the number of bugs fixed (especially semantic and logical) was significantly higher when written notes were available. Since no differences in the strategies used between the groups were recognizable, the authors concluded that the differences had to be on a higher level and, above all, that a systematic process to debugging was decisive.

Böttcher et al. [6] provided a systematic debugging procedure (see figure 1) and the use of the debugger in an explicit teaching unit. The debugging procedure – using the Wolf Fence algorithm (binary search) as an analogy – was explained in a live demonstration and an exercise with debugging tasks was performed. The evaluation

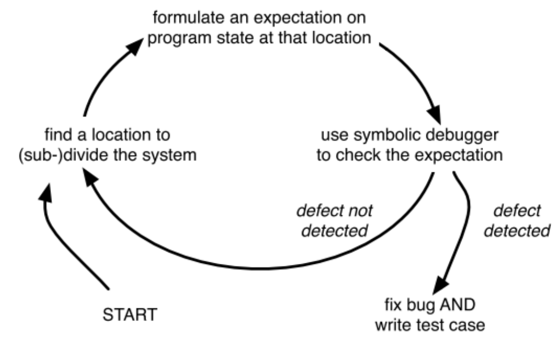
formulate an expectation on program state at that location
use symbolic debugger to check the expectation
find a location to (sub-)divide the system
defect not detected
defect detected
START
fix bug AND write test case

Figure 1: Debugging procedure conveyed by Böttcher et al. [6]

showed that only a few students applied the conveyed systematic process, but quickly returned to a non-systematic "poking around".

2.3.2 Debugging in the K12 classroom. Carver and Risinger [8] provided a systematic debugging process for LOGO with promising results: They gave the students one hour of debugging training as part of a larger LOGO curriculum. They used a flow chart that characterized the debugging process (see figure 2), "bug mappings", and debugging logs that were available in the classroom. The results (without a control group) showed a shift from brute force to a systematic approach.

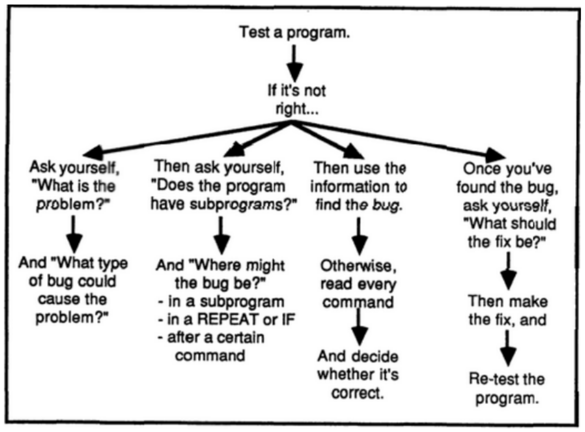
Test a program.
If it's not right...
Ask yourself, "What is the problem?"
And "What type of bug could cause the problem?"
Then ask yourself, "Does the program have subprograms?"
And "Where might the bug be?" - in a subprogram - in a REPEAT or IF - after a certain command
Then use the information to find the bug.
Otherwise, read every command
And decide whether it's correct.
Once you've found the bug, ask yourself, "What should the fix be?"
Then make the fix, and
Re-test the program.

Figure 2: Step-by-step debugging procedure conveyed by Carver and Risinger [8]

In summary, those findings show that conveying a systematic debugging process seems particularly promising for improving students' debugging skills. However, teaching a systematic debugging process hardly plays a role in the K12 classroom, where the unsystematic teaching of debugging strategies or tools prevails [24]. Overall, an up-to-date and methodically sound study investigating the effect of explicitly teaching a systematic process is lacking.

3 METHODOLOGY

3.1 Research Questions

The aim of this study is to investigate the influence of explicit teaching of a systematic process to debugging in class.

Fostering self-reliance in debugging in particular seems to be an important step towards enabling students to autonomously cope with their errors. Therefore, we are interested in the students' perceived self-efficacy, as introduced by Bandura [5]. Ultimately, the underlying goal of teaching debugging is to improve the actual debugging performance, i.e. increase the numbers of errors fixed.

To this end, we will address the following research questions:

- (**RQ1**) Does teaching a systematic debugging process have a positive effect on students' debugging self-efficacy?
- (**RQ2**) Does teaching a systematic debugging process have a positive effect on students' debugging performance?

3.2 Study Design

To answer these research questions, we have chosen a pre-post-control-group-test-design. First, the intervention was piloted without a control group in a year ten (ages 15 to 16) class for particularly high performing students ($n = 14$, using Greenfoot and Stride in the classroom) to make adjustments based on the findings of this implementation. Results from such a study without a control group, however, help us to answer the research questions only to a limited extent since possible increases in self-efficacy expectations and student performance could also be attributed to the additional exercise in debugging. In order to investigate the influence of the intervention in contrast to pure debugging practice, e.g. through debugging tasks, we used two year ten classes, the first as an experimental ($n = 13$), the second as a control group ($n = 15$). We intentionally selected two classes that were taught by the same teacher with an identical teaching concept (using BlueJ and Java), and were equally advanced in the curriculum at the time of our study.

Each 90-minute lesson – led by the first author – consisted of a pre-test, an intervention of about 10 minutes (except for the control group) and a post-test. As shown in figure 3, the pre- and post-tests were divided into a questionnaire to assess self-efficacy expectations (four items with a five-level Likert scale) and the solvability of the tasks (only in the post-test) as well as debugging tasks to assess the students' performance. Regarding the latter, the number of corrected errors (analogous to [12]) was used as measurement. For this, both the worksheets, on which errors and their corrections had to be noted by all groups, and the code were evaluated.

3.3 Intervention

The intervention provides a systematic process to finding and fixing errors, according to the causal reasoning of the "scientific method" [36]: based on the observed behavior of the program, repeated hypotheses are formulated, verified in experiments and, if necessary, refined until the cause is found. We use a didactically adapted variant of this procedure and explicitly distinguish between different error types – compile time, runtime and logical errors (see figure 4) – and reflect on the hierarchy in fixing these errors, as the debugging process varies greatly depending on the underlying error type. The *undoing* of changes is emphasized if troubleshooting measures

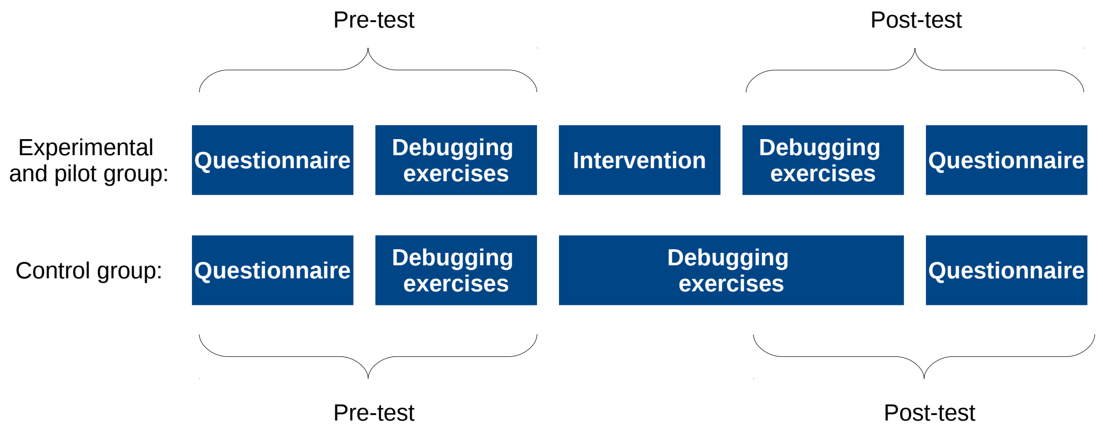

Figure 3: Study design

are not successful – especially since this procedure is unnatural for students [30]. This is to prevent students from incorporating additional bugs during failed troubleshooting – a typical phenomenon for novice programmers [15].

Carrying out the intervention, the systematic debugging process was presented on a poster (see figure 4), discussed with the students and stepped through for an example. The poster was displayed in the classroom for the remaining lesson. In the post test, the material for the experimental and pilot group reminded the students to apply the previously conveyed approach.

3.4 Debugging exercises

The challenge in this study was to measure the actual debugging performance. Debugging exercises are typically used to practice and assess debugging. However, these confront students with a large amount of foreign code. Katz and Anderson [19] found that the procedure for debugging your own programs differs from that of other programs. Furthermore, understanding and getting used to foreign code is a big challenge for novices, as competencies such as program comprehension and tracing are paramount, as Allwood points out [2]. In order to approach the actual goal, improving the debugging of *own* programs and investigating of distinct debugging skills we therefore use several prototypes of a program that build on each other. In this way, the students in each new prototype are confronted with comparatively little "foreign" code and are already familiar with the "old" code. For example, in the first prototype of the pong game used in the pilot group only the movement of the ball is implemented, and in the next one, the clubs and their controls are additionally inserted.

Since the debugging and not the test skills of the students were to be examined, the number of errors per prototype was given. For the same reason, care was taken to ensure that the malfunction of the program was quickly apparent so that the error localization could be started directly. I.e., there was no need to design edge cases or malformed input to be able to observe erroneous program behavior. The bugs encountered were common syntactical (e. g. missing braces or data types), run-time (e. g. missing initialization that leads to run-time exceptions) as well as logical errors (e. g. missing method calls or interchanged movement directions).

4 RESULTS

4.1 (RQ1) Does teaching a systematic debugging process have a positive effect on students' debugging self-efficacy?

First we examined the change of self-efficacy expectations for the pilot, experimental and control groups, both pre and post, which results from the mean value of the four items assessed in the respective questionnaires. The answers of the five-level Likert scale were mapped on a scale of 0 (do not agree) to 4 (agree). The mean values therefore range between 0 and 4.

We determined whether there is a significant change of self-efficacy expectations between pre- and post-test within the individual groups. Due to the sample sizes, we always used non-parametric methods for testing significance [29]. Using the Wilcoxon signed-rank test – a non-parametric test for dependent samples – we analyzed the rankings in pre- and post-tests. In table 1 the respective medians and the p-value of the Wilcoxon signed-rank test (H_0: no or negative test effect) are shown [1].

	Median pre	Median post	Wilcoxon test
Pilot group	2.75	3.25	$p = 0.044^*$
Control group	2.25	2.50	$p = 0.083$
Experimental group	2.25	2.75	$p = 0.001^*$

Table 1: Influence on self-efficacy expectations

[1]Significant test results at a significance level of $\alpha = 0.05$ are marked by a $*$.

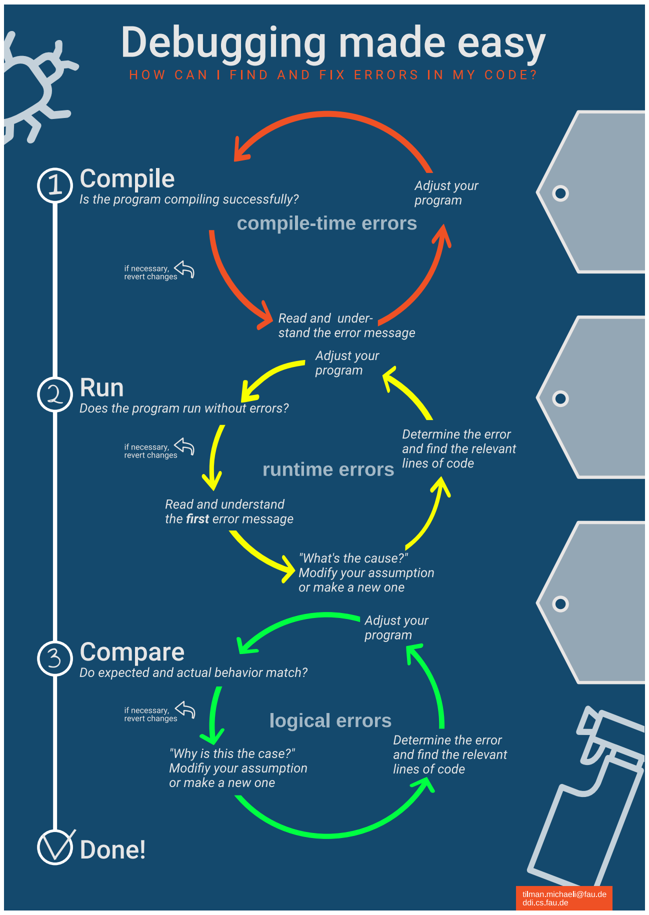
Debugging made easy
HOW CAN I FIND AND FIX ERRORS IN MY CODE?
1 Compile
Is the program compiling successfully?
Adjust your program
compile-time errors
if necessary, revert changes
Read and understand the error message
2 Run
Does the program run without errors?
Adjust your program
Determine the error and find the relevant lines of code
if necessary, revert changes
runtime errors
Read and understand the first error message
"What's the cause?" Modify your assumption or make a new one
3 Compare
Do expected and actual behavior match?
Adjust your program
if necessary, revert changes
logical errors
"Why is this the case?" Modify your assumption or make a new one
Determine the error and find the relevant lines of code
Done!
tilman.michaeli@fau.de
ddi.cs.fau.de

Figure 4: Systematic debugging process conveyed in the intervention

We see an increase in self-efficacy expectations in all three groups. However, this is only significant for the pilot and experimental groups at a significance level of $\alpha = 0.05$. The effect sizes according to Cohen are $d = 0.56$ (pilot) and $d = 0.54$ (experimental), which corresponds to a medium effect [10].

Although active debugging improves self-efficacy expectations, a systematic process seems to have a stronger impact on self-efficacy expectations.

4.2 (RQ2) Does teaching a systematic debugging process have a positive effect on the debugging performance of the students?

For differences in debugging performance, we compared the performances of the experimental and control groups in pre- and post-tests. Debugging performance is measured by the number of bugs fixed. A pre-post comparison of the debugging performance within the individual groups – as it was done for *RQ1* – is not applicable, since different errors had to be corrected in pre- and post-test.

To determine the effect of the intervention, we have to compare the students' debugging performance pre and post. Therefore, we had to examine whether the two samples came from the same population at both points of time. We can only assume a significant increase between the groups' performance when they are from the same population pre, but not post. Here, too, we used a non-parametric test, the Mann-Whitney-U-test. In contrast to the Wilcoxon signed-rank test, this test is designed for independent samples. The p values of the Mann-Whitney-U test (H_0: samples come from the same population) are shown in table 2.

	Mann-Whitney-U-Test
Experimental vs. control group Pre	$p = 0.191$
Experimental vs. control group Post	$p = 0.049^*$

Table 2: Influence on debugging performance

Accordingly, we cannot reject the null hypothesis for comparing pre-tests at a significance level of $\alpha = 0.05$: The debugging performance of the students did not differ significantly before the intervention. In contrast, there is a significant difference in the post-test: The students in the experimental group had a higher debugging performance (median = 4, with a total of 9 errors to be corrected) than the students in the control group (median = 2). In the post-test, tasks with a higher degree of difficulty were used to determine the debugging performance, since a learning effect between the pre- and post-test can be assumed in both groups. The effect size according to Cohen is $d = 0.69$ and corresponds to a medium effect [10].

The higher debugging performance is also reflected in the perceived difficulty of the tasks by the students. This was determined ex post in the questionnaire using a five-level Likert scale. Using the same scale as before, 0 (do not agree) to 4 (agree), results in the following mean values:

	Tasks Pre	Tasks Post
Control group	3.07	1.47
Experimental group	3.23	2.92

Table 3: Mean values for *"The tasks could be easily solved"*

The results suggest that a systematic process can make the difference: if students are given such a systematic process, they can significantly improve their success in localizing and correcting errors.

5 DISCUSSION

The aim of this intervention study was to examine the influence of explicit teaching of a systematic process to debugging and its influence on students' *self-efficacy* and *debugging performance.*

In line with the conclusions that McCauley et al. [23] draw from their extensive literature review, the intervention focuses on competencies involved in pursuing a systematic approach to debugging. This involves competencies such as *reasoning about the program* *based on (erroneous) output* and *formulating and modifying hypotheses.*

Because this intervention covers findings from further research regarding novice programmers such as *getting stuck* [12] or the tendency to *introduce new bugs* [15], this intervention goes beyond similar studies discussed in the section on related work. Furthermore, in contrast to studies that predominantly focus on programs that already compile (c.f. [6]) in university settings, we take different error types into consideration. This seems important, as for example especially *compile-time errors* pose a big hurdle in the K12 classroom [24].

A typical phenomenon in the K12 classroom is the teacher hurrying from one student to the other (c.f. [24]), explaining errors and giving hints for troubleshooting. Especially since the concept of "learned helplessness" [28] might play a role in this [24], fostering students' debugging self-efficacy and eventually self-reliance is essential. Our approach is designed to foster the *self-reliance* of students in debugging by giving explicit instructions on what to do if they encounter errors. Additionally, our goal was to tackle the trial-and-error approach, which is common especially for "weaker" students [25], by fostering a planned and deliberate process by calling for them to explicitly formulate hypotheses. Our results support that these goals have been met.

Although we have deliberately chosen a classroom setting over a lab setting for our study and, therefore, have no data regarding the students' actual debugging process and possible changes after the intervention, our results and the unstructured observations from the classrooms show, that indeed, students successfully applied a systematic approach that eventually lead to a increase in self-efficacy and a higher debugging performance.

During our intervention, we observed that the students were not aware of the existence of different classes of errors, despite already having been confronted with them on a regular basis. Only explicit reflection on the errors they encountered within our intervention made the students realize the differences in how they came into contact with them, which information the IDE provided, and how they dealt with the different errors.

We have tested our systematic process with both Java and BlueJ (experimental and control group) as well as with Stride and Greenfoot (pilot). The positive results in both cases indicate that this approach is independent of tools and (text-based) programming languages (in contrary to e.g. [8]). This makes it suitable for the classroom, where a great heterogeneity in the tools and languages used prevails.

What significance do these results have for computer science teaching? According to [24], computer science teachers lack suitable concepts for teaching *debugging*: although some unsystematic debugging strategies, as well as the use of tools like the debugger, are occasionally the subject of teaching, the teaching of a systematic procedure has hardly played a role so far. This study underlines how important it is to convey such a systematic debugging process and offers an hands-on approach for the classroom.

5.1 Threats to Validity

A possible limitation of the validity of this study is the small sample size and the lack of randomization of students. They were taught by the same teacher according to the same concept and also came from the same school. This could limit the significance of the results when generalizing to the population as a whole. We therefore plan to extend this study to a larger sample.

6 CONCLUSION

In this study, we examined the influence of the explicit teaching of a systematic debugging process regarding the influence on students' self-efficacy and debugging performance. To this end, we developed an intervention, piloted it and then examined it in a pre-post-control-group-test-design: Both the experimental and control group were surveyed by a questionnaire and given debugging exercises as a pre-test. Afterwards, in the experimental group, the intervention was carried out while the control group continued to work on debugging exercises. As post-test, the students once more worked on debugging exercises and were surveyed.

Our data shows that such an intervention is a promising approach to teaching debugging skills:

- The teaching of a systematic process for finding and correcting programming errors has a positive impact on debugging self-efficacy.
- Students who have been taught a systematic process also perform better in debugging than students who have practiced debugging exclusively.

The presented intervention represents a first building block for the promotion of debugging skills. This should be extended by teaching concrete debugging strategies, the usage of debugging tools and building up heuristics and patterns. In summary, our study provides empirical arguments for explicitly teaching debugging and simultaneously offers an hands-on approach for the classroom.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice Computer Science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '05)* 37, 3 (2005), 84–88.
[2] Carl Martin Allwood and Carl-Gustav Björhag. 1990. Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies* 33, 6 (1990), 707–724.
[3] Carl Martin Allwood and Carl-Gustav Björhag. 1991. Training of Pascal novices' error handling ability. *Acta Psychologica* 78, 1-3 (1991), 137–150.
[4] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.* ACM, New York, NY, USA, 522–527.
[5] Albert Bandura. 1982. Self-efficacy mechanism in human agency. *American psychologist* 37, 2 (1982), 122.
[6] Axel Böttcher, Veronika Thurner, Kathrin Schlierkamp, and Daniela Zehetmeier. 2016. Debugging students' debugging process. In *2016 IEEE Frontiers in Education Conference (FIE).* IEEE, Erie, PA, USA, 1–7.
[7] Neil CC Brown, Sue Sentance, Tom Crick, and Simon Humphreys. 2014. Restart: The resurgence of computer science in UK schools. *ACM Transactions on Computing Education (TOCE)* 14, 2 (2014), 9.
[8] McCoy Sharon Carver and Sally Clarke Risinger. 1987. Improving children's debugging skills. In *Empirical studies of programmers: Second workshop.* Ablex Publishing Corp., Norwood, NJ, USA, 147–171.
[9] Ryan Chmiel and Michael C Loui. 2004. Debugging: from Novice to Expert. *Proceedings of the 35th SIGCSE technical symposium on Computer science education - SIGCSE '04* 36, 1 (2004), 17.
[10] Jacob Cohen. 1988. *Statistical power analysis for the behavioural sciences.* Hillsdale, NJ: erlbaum, New York, NY, USA.
[11] Mireille Ducasse and A-M Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering.* IEEE Computer Society Press, Piscataway, NJ, USA, 162–171.

[12] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[13] David J Gilmore. 1991. Models of debugging. *Acta psychologica* 78, 1-3 (1991), 151–172.

[14] John D. Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182.

[15] Leo Gugerty and G. Olson. 1986. Debugging by skilled and novice programmers. *ACM SIGCHI Bulletin* 17, 4 (1986), 171–174.

[16] Morgan Hall, Keri Laughter, Jessica Brown, Chelynn Day, Christopher Thatcher, and Renee Bryce. 2012. An empirical study of programming bugs in CS1, CS2, and CS3 homework submissions. *Journal of Computing Sciences in Colleges* 28, 2 (2012), 87–94.

[17] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin* 35, 1 (2003), 153–156.

[18] W Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. *Bug catalogue: I.* Yale University Press, New Haven, CT, USA.

[19] Irvin R. Katz and John R. Anderson. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[20] Claudius M Kessler and John R Anderson. 1986. A model of novice debugging in LISP. In *Proceedings of the First Workshop on Empirical Studies of Programmers.* Ablex Publishing Corp., Norwood, NJ, USA, 198–212.

[21] Andrew J Ko and Brad A Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1-2 (2005), 41–84.

[22] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *Acm Sigcse Bulletin* 37, 3 (2005), 14–18.

[23] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[24] Tilman Michaeli and Ralf Romeike. 2019. Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. In *2019 IEEE Global Engineering Education Conference (EDUCON).* IEEE, Dubai, VAE, 1030–1038.

[25] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin.* ACM, Portland, OR, USA, 163–167.

[26] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of learning in novice programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55.

[27] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110.

[28] Christopher Peterson, Steven F Maier, and Martin E P Seligman. 1993. *Learned helplessness: A theory for the age of personal control.* Oxford University Press, New York, NY, USA.

[29] Björn Rasch, Malte Friese, Wilhelm Hofmann, and Ewald Naumann. 2010. Quantitative Methoden 2: Einführung in die Statistik für Psychologen und Sozialwissenschaftler (3., erweiterte Auflage). (2010).

[30] Beth Simon, Dennis Bouvier, Tzu-yi Chen, Gary Lewandowski, Robert Mccartney, and Kate Sanders. 2008. Common sense computing (episode 4): debugging. *Computer Science Education* 18, 2 (2008), 117–133.

[31] Diomidis Spinellis. 2018. Modern debugging: the art of finding a needle in a haystack. *Commun. ACM* 61, 11 (2018), 124–134.

[32] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[33] James G Spohrer and Elliot Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers.* Ablex Publishing Corp., Norwood, NJ, USA, 230–251.

[34] Iris Vessey. 1985. Expertise in Debugging Computer Programs: Situation-Based versus Model-Based Problem Solving. *International Conference on Information Systems (ICIS)* 18 (1985), 18.

[35] Aman Yadav, Ninger Zhou, Chris Mayfield, Susanne Hambrusch, and John T Korb. 2011. Introducing Computational Thinking in Education Courses. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11).* ACM, New York, NY, USA, 465–470.

[36] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann, Burlington, MA, USA.