Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training

Luisa Greifenstein  
University of Passau  
Passau, Germany

Florian Obermüller  
University of Passau  
Passau, Germany

Ewald Wasmeier  
University of Passau  
Passau, Germany

Ute Heuer  
University of Passau  
Passau, Germany

Gordon Fraser  
University of Passau  
Passau, Germany

KEYWORDS
Scratch, Block-based programming, Bug patterns, Hints, Teachers

1 INTRODUCTION

Block-based programming is frequently used as entry-point to computational thinking and to programming, since the block-based nature reduces complexity compared to text-based programming languages. Nevertheless, there are many concepts to comprehend, and misconceptions about these concepts may hamper progress in learning. These misconceptions often manifest in “buggy” programs, i.e., programs that contain mistakes and do not function correctly. It is then up to the teacher to help learners identify these bugs in their programs, and to fix or explain them in order to overcome the misconceptions that caused them. While subject teachers at secondary schools may be expected to be adequately educated in debugging, programming is increasingly also introduced at primary schools, where teachers are not specialised in all individual subjects they teach and thus may not have adequate training.

Teachers have the advantage that learners frequently have similar misconceptions, which result in recurring patterns of bugs in their programs. For example, a common misconception of learners is that an if-statement triggers each time the condition becomes true [26]. Figure 1a shows a code snippet from a Scratch [14] program where this misconception manifests in a bug. A correct implementation (Figure 1b) would enclose the check inside a forever-loop to continuously check for the occurrence of the event. Bugs matching this pattern occur frequently: In a recent study, this Missing Loop Sensing bug pattern was found in 3,282 projects in a random dataset of 74,830 Scratch projects [5].

Instances of bug patterns like the Missing Loop Sensing example above can be detected automatically. The repetitive nature of these bug patterns also makes it possible to provide generic hints on how to fix the identified bugs, and what misconceptions may cause them. However, especially primary school students might be overwhelmed by the concepts and the technical terminology used...
Greifenstein et al.

would expect that an appropriate hint pointing out the location of the primary school teachers in training. We first compared the differences between a subgroup of our participants who were given hints on bug patterns (Group \textit{Trmt}) versus a subgroup not receiving any hints (Group \textit{Ctrl}). This allows us to quantify effects on their success in fixing the bugs as well as the time it took them to do so. In order to evaluate which learning experiences can be transferred better, we then experimentally determined whether the participants of Group \textit{Ctrl} were able to spot similar bugs in other programs more easily without the hints than Group \textit{Trmt}. This provides us with information whether hints can be transferred, or whether missing the learning experience of manually identifying the bugs hampers the debugging skills of teachers in training.

Our experiment confirms that giving hints on bug patterns improves the performance at correcting bugs described by the hints: In general, hints can reduce the effort of finding and fixing bugs in terms of time from 8.66 to 5.24 minutes, while increasing the effectiveness in terms of correct programs by 34\% more correct solutions. This suggests that providing tool-support to recognise bug patterns is desirable for teachers in practice, and research on identifying bug patterns has the potential to improve classroom learning. We do, however, observe that the beneficial effects are larger for participants having prior programming experience. Although our experiment provides no significant evidence that either learning to debug with hints or learning to debug “the hard way” leads to better learning effects, the knowledge of the bug patterns that was gained by the hints seems to slightly improve performance on similar tasks without hints. This suggests that bug patterns might be a useful concept to include in the curriculum for teachers in training.

\section{2 RELATED WORK}

\subsection{2.1 Misconceptions and Bug Patterns}

Learners of programming can have misconceptions about programming constructs \cite{28}. Such misconceptions may severely hamper their ability to progress, as they will lead to programs that do not work and are a source of frustration \cite{8}. In the context of beginner programming, Sorva \cite{26} identified a catalogue of 41 such common misconceptions; for example, the misconception that variables can store multiple values and store the history of values assigned to it, or that a while loop’s condition is evaluated continuously and the loop exits the instant it becomes false. Common misconceptions may result in similarly erroneous (i.e., buggy) programs. Even though programs may be functionally very different, the bugs resulting from misconceptions may be very similar. Fraedrich et al. \cite{5} introduced the notion of common patterns of such bugs. Generally speaking, a bug pattern in a block-based programming language is

\begin{figure}[ht]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{An example of the \textit{Missing Loop Sensing} bug pattern: Instead of continuously checking for collisions with a \textit{fish} sprite, the buggy snippet performs just a single check immediately after initialising the variable \textit{points}—this will scarcely suffice to detect all possible touching events.}
\end{figure}

\begin{figure}[ht]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Example hint provided by \textit{LitterBox} to support recognising and fixing the \textit{Missing Clone Initialisation} bug pattern present in Task 4.}
\end{figure}
a composition of blocks that is typical of defective code [5]. Such bug patterns can be observed frequently in practice: Fraedrich et al. [5] found that out of a sample of 74,830 publicly shared Scratch projects, 33,655 contained at least one instance of a bug pattern.

Of course, not all bugs result in bug patterns. As bug patterns do not deal with task-specific information but the code itself, bugs such as using the wrong variable, the wrong content in e.g. say blocks or forgetting to implement certain features cannot be detected. Such content-related bugs can be addressed with dynamic analysis tools for Scratch such as Whisker [27]. They run the code and check if specified events occur or not. This implies that the intended aim of the program must be known and respective tests must be available or created by the teacher. This is why dynamic testing cannot be easily applied to programs that result from open programming tasks and without feeding the tool with further individualised information, no feedback on how to proceed can be provided (apart from that the feature must be changed or implemented).

In contrast to that, static analysis tools enable teachers and students to analyse programs without having to specify its intended output. Furthermore, when such tools check if instances of bug patterns exist they can provide generic information on the detected bug patterns, such as knowledge on the associated misconception or on how to remove the respective bug pattern. This is done by the LitterBox [4] tool as it can be seen in the example hint in Fig. 2. Intuitively, such knowledge can aid in the process of determining whether code is correct, or in the process of debugging defective code: If an instance of a bug pattern has been spotted, this is a suitable point to start debugging. Indeed, bug patterns tend to lead to functionally defective programs: Out of 250 manually analysed projects 218 contained bug patterns that result in erroneous behaviour [5]. Other existing code analysis tools for Scratch such as Hairball [3], Quality Hound [29] or Dr. Scratch [18] report general quality problems in terms of code smells (= unaesthetic or less readable code that does not harm the functionality of a program), rather than specific bug patterns resulting from misconceptions. Previous research has shown that these code smells in Scratch hamper novices’ learning processes [9]. Bug patterns are more severe than code smells, thus likely having an even worse impact on novice programmers; our investigations in this paper suggest a possible way of dealing with them.

### 2.2 Feedback on Code

The ability to detect such bug patterns in student code makes it possible to provide learners and teachers with feedback. Kennedy et al. [11] demonstrated—using peer feedback and discussions of coding assignments—that misconceptions can be cleared with feedback in principle. In this paper, we evaluate the effects of feedback that is generated automatically by static analysis. Past research on automated hint generation has mainly considered the problem of providing hints on what should be the next step in solving programming assignments [20, 25, 30, 32] or open ended programming tasks [19, 21] and how novices seek help in these systems [1, 15, 16, 22]. An exception is the work by Gusukuma et al. [6], who showed that feedback delivery on mistakes that anticipate possible misconceptions generally leads to favourable results, and that showing such hints does not harm transfer to new tasks. In contrast to this work, the hints we evaluate are not tailored to our self-study material and thus might be more universally applicable.

### 2.3 Teachers as Debuggers

Not only students but also teachers—who are expected to support their students—struggle with debugging. Sentance and Csizmadia [24] examined the opinions of teachers in the UK about challenges when teaching computing. The most frequently mentioned challenge was the teacher’s subject knowledge, and in particular this was mentioned more often by primary school teachers than by secondary school teachers. Within this challenge, teachers were concerned if they were able to help their students with their problems when programming. Yadav et al. [31] came to similar conclusions after interviewing high school computer science teachers, where they identified both content and pedagogical challenges. Supporting students when teaching programming is perceived as a main difficulty—not only because of the partially missing computer science background: The teachers explained this challenge also with (1) the teacher-student ratio and (2) the different programming approaches of the students that result in different needs. Teachers’ debugging performance has been studied explicitly by Kim et al. [12], who examined what kind of bugs early childhood preschool teachers produce and how they deal with given bugs: The teachers had difficulties debugging given programs, even though they tried different debugging strategies. Michaeli and Romeike [17] interviewed high school teachers on their strategies to cope with programming bugs of their students. Most teachers reported that they have to help efficiently to reach all students with problems and that there is no time for detailed explanations. Consequently, there is a need to examine how teachers—especially those with insufficient subject knowledge—can be supported in debugging. The aim of this paper is to generate an initial understanding of the effects of hints on bug patterns. Do such hints have a positive influence on the effectiveness, efficiency and learning opportunities of teachers inspecting their students’ programs?

### 3 STUDY SETUP

To evaluate the impact of hints on bug patterns on the performance of teachers, we aim to answer the following research questions:

**RQ 1:** How do hints on bug patterns influence the effectiveness and efficiency of teachers while debugging and fixing bugs?

**RQ 2:** How do hints on bug patterns influence the ability of teachers in training to recognise and fix bugs without hints?

To answer these research questions, we conducted an A/B study with primary school teachers in training, who were tasked to fix several buggy Scratch programs, with and without hints.
We implemented a three-week programming session into a course with 242 students signed up at the University of Passau. We chose previous tasks. Additionally, we excluded 11 participants of Group Ctrl and 3 participants of Group Trmt because some of their submitted programs were corrupted or because they submitted programs of other tasks. We excluded 5 participants of Group Ctrl and 7 participants of Group Trmt because they did not submit all tasks. This allows us to compare the performance across tasks, which would be challenging if participants would differ between tasks.

3.1 Study Participants

We implemented a three-week programming session into a course on mathematical didactics for primary schools teachers in training with 242 students signed up at the University of Passau. We chose the participants of this course because primary school teachers have to teach almost every subject but they are not specialised in all of them. Thus, they may not have had adequate training and may lack adequate skills for debugging. Although computer science is not yet part of the Bavarian curriculum, programming is increasingly introduced in primary schools.

While course participants were recommended to take part in the programming sessions, it was not mandatory as they had a choice of which sessions of the overall course their assessment should be based on. Of the registered students, 189 actively participated, but when participants did not submit all tasks, submitted corrupted files, or submitted programs for the wrong tasks, we conservatively excluded all data of these participants. Overall, we excluded all data points of 26 participants. We excluded 5 participants of Group Ctrl and 3 participants of Group Trmt because some of their submitted programs were corrupted or because they submitted programs of previous tasks. Additionally, we excluded 11 participants of Group Ctrl and 7 participants of Group Trmt because they did not submit all tasks. This

| Table 1: Bug patterns and corresponding hints. |
|----------------------------------------------|
| **Bug Pattern**                             | **LITTERBox Hint**                                                                 |
| Message Never Sent (Tasks 1 and 8)          | The message “Game Over”, that is to be received here, is never sent. Therefore, the adjacent script will never be triggered. If you want to receive a message, you have to select a message, that is already sent via a different script or you have to create and send a matching message inside a different script. |
| Missing Loop Sensing (Tasks 2 and 9)        | The highlighted condition is checked only once. Thus, the script runs through too fast. Implement a continuous check by surrounding the conditional statement with a forever loop. |
| Comparing Literals (Tasks 3 and 10)         | You used current year just as plain text. Your comparison will always return FALSE. Therefore, the code following wait until is never executed. Did you rather intend to use your already existing variable block current year instead of the plain text? You will find this block in the Variables toolbox. |
| Missing Clone Initialization (Tasks 4 and 11) | When you clone a sprite that has no script beginning with When I start as a clone or When this sprite clicked, the clone is unable to do some work. You are using a delete this clone block in a script of Sprite Red. Consider using a different event handler block like When I start as a clone or When this sprite clicked at the beginning of this script. |
| Message Never Received (Tasks 5 and 12)     | The message "player touches money" that is sent here is never received by a when I receive "player touches money" block. Therefore nothing will happen as a reaction to this message. When you send a message make sure that another script receives it. |
| Forever Inside Loop (Tasks 6 and 13)        | The inner forever loop is never left. Therefore, all blocks in this script but outside of the forever loop are never executed again. Try omitting the inner forever loop. |
| Stuttering Movement (Tasks 7 and 14)        | If you continuously press a key, you expect a smooth event processing. Unfortunately, a delay occurs between the first and the second processing round, resulting in stuttering movement. You can prevent this delay to happen by using the key right arrow pressed? block from the Sensing toolbox. To do that, you have to put the conditional statement if key right arrow pressed? then inside of a forever loop and use the event handler When green flag clicked instead of the event handler When key right arrow pressed. |

This results in a total of 163 participants, of which 142 were female, 21 were male, and they were between 18 and 45 years of age with a median of 20 years. According to our demographic pre-survey, many participants had no prior experience in programming at all, which means they neither programmed at school nor at university or anywhere else (Group Ctrl: 27%; Group Trmt: 32%). We will refer to these participants as inexperienced participants and to those with some prior experience as experienced participants. To ensure all participants had sufficient basic knowledge for the experiment, we provided self-study material that can be completed in 90 minutes: Within one week, the participants were tasked to watch a 30 minutes explanatory video and perform three exercises. In the video a SCRATCH program is built step by step, thereby covering all blocks and concepts that were needed to fix the buggy SCRATCH programs later in the experiment. Specific misconceptions and bug patterns were not discussed. The participants had to stop the video three times to perform an exercise. This approach is based on the Use-Modify-Create-Framework [13]: At first, they used the presented program by opening, running and interpreting it. In the second exercise, they expanded the functionality of one sprite. Finally, they created their own program in about 30 minutes.
3.2 Experiment Procedure

3.2.1 Overview. After introducing the participants to Scratch with the self-study material (described in Section 3.1), the main part of the experiment started: The participants were tasked to fix one example program (Task 0), which served to familiarise participants with the infrastructure, and 14 buggy Scratch programs (Tasks 1–14) within two weeks using their own digital devices. They should spend about ten minutes on one task but were allowed to go on to the next task, regardless of the current task being completed or not. However, once they had proceeded to the next task, they were not able to go back to previous tasks. The tasks were provided on a website created by the researchers. The participants downloaded the broken programs, edited them on the public Scratch website and then again uploaded the edited programs to our website—even if they could not fix it. The participants were asked to pause only at dedicated pages (after each task) to get meaningful information about the time on task.

3.2.2 Scenario. For each task a fictitious primary school student describes his or her program. To ensure that our results cannot be attributed to differences in the descriptions, the 14 descriptions are structured and filled with content in the same way (an example is shown in Fig. 3). In the first two paragraphs, the students says hello and that he or she wants to create an animation or a game. In the following paragraphs the student explains which functions have already been correctly implemented. In the last paragraph the student describes what does not work yet and thus the desired final state. After having repaired and submitted the given program the participants wrote feedback on the bug to the fictitious primary school student. Giving feedback corresponds to the procedure in school and we intended that the teachers in training reflected more on the bug: We attempted that participants repair the program less by trial and error or by following the hint passively but are rather cognitively strongly involved. When participants were not able to repair a program, they still uploaded their final result and instead of feedback on the bug, they were allowed to write feedback on a positive implementation in the student’s program instead.

3.2.3 A/B study. While course participants were recommended to take part in the programming sessions, it was not mandatory. We therefore assigned participants to Group Ctrl or Group Trmt on-the-fly through the web interface: Whenever a participant started with Task 1, they were assigned to the smaller of the two groups. The example task (Task 0) was identical for both groups. The first half of the following tasks (Tasks 1–7) differed between Group Ctrl and Group Trmt only in that Group Trmt received one hint for each task. The second half of the tasks (Tasks 8–14) did not differ between the groups: no hints were given for either group. In this way we can examine if there are any effects of using hints within the first tasks and then not providing them within the following tasks that contain the same bug patterns but no hints.

3.2.4 Bug Patterns, Hints and Tasks. For each task, we created or adapted a Scratch program based on projects used previously for teaching. Table 2 shows the values of different complexity measures for each program. We modified each of these programs to contain one instance of a bug pattern.

We decided to use the seven most common bug patterns described by Fraidrich et al. [5], who found each of these bug patterns in more than 1,500 Scratch projects of 33,655 publicly shared buggy Scratch projects. By choosing exactly seven bug patterns we made a compromise between a reasonable amount of time and the largest number of bug patterns and thus generalisability. We used the bug patterns in the order shown in Table 1. By this, we attempted to maximise the temporal distance between bug patterns which are related—such as Message Never Sent and Message Never Received or Missing Loop Sensing and Forever Inside Loop.

The hints given to Group Trmt for Tasks 1 to 7 were generated using LitterBox. As shown in Fig. 2, each hint contains both images and text. The text consists of (1) an explanation of the problem and a clarification of the underlying misconception and (2) a generic suggestion on how to remove the bug pattern. The images show the sprite and the script with the highlighted block, where the cause of the bug is located. Table 2 also lists the hints for Tasks 1 to 7. As LitterBox analyses the code statically and does not deal with content information about the programming task, some hints provide several options on how to proceed. For the bug pattern Missing Clone Initialisation (Fig. 2), e.g., two different event handlers could solve the bug pattern but only one of them might make sense in the associated program. Thus, the user has to decide which option to use to achieve the intended aim of the program. Consequently, it must be examined to what extent generic hints on bug patterns can support learners.

### Table 2: Complexity of the programs.

| Task | Blocks | Scripts | Sprites | HD | HE | HL | HS | HV | ICC | WMC |
|------|--------|---------|---------|----|----|----|----|----|-----|-----|
| 1    | 81     | 10      | 3       | 3.9 | 1.3 | 1.8 | 1.7 | 1.7 | 1.8 | 1.9 |
| 2    | 44     | 6       | 2       | 2.0 | 1.7 | 1.4 | 1.4 | 1.4 | 1.5 | 1.5 |
| 3    | 34     | 5       | 2       | 4.7 | 4.7 | 4.7 | 4.7 | 4.7 | 4.7 | 4.7 |
| 4    | 52     | 6       | 2       | 2.8 | 2.8 | 2.8 | 2.8 | 2.8 | 2.8 | 2.8 |
| 5    | 47     | 7       | 4       | 2.4 | 2.4 | 2.4 | 2.4 | 2.4 | 2.4 | 2.4 |
| 6    | 36     | 5       | 4       | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 |
| 7    | 62     | 9       | 4       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 8    | 56     | 10      | 5       | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 | 1.6 |
| 9    | 49     | 6       | 4       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 10   | 66     | 8       | 6       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 11   | 77     | 13      | 4       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 12   | 61     | 7       | 2       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 13   | 38     | 3       | 2       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| 14   | 39     | 7       | 2       | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |

1Halstead Difficulty [7] 2Halstead Effort [7] 3Halstead Length [7] 4Halstead Size [7] 5Halstead Volume [7] 6Interprocedural Cyclomatic Complexity 7Weighted Method Count

3.3 Data Analysis

We analysed the participants’ performance using the criteria (1) correct fixing of the broken functionalities and (2) time needed.

3.3.1 Effectiveness. We used a Chi-squared test to measure if the functionality differs significantly between Group Trmt and Group Ctrl and used odds ratio (OR) to calculate the effect size. If OR=1.0, there are no effects in favour of any group. Values below 1.0

---

1http://scratch-litterbox.org/, last accessed 27.05.2021
indicate effects in favour of Group Ctrl and values over 1.0 indicate effects in favour of Group Trmt.

To find out if a broken functionality was correctly fixed we mainly used automated Whisker [27] tests. Whisker tests Scratch programs dynamically, which means that Whisker runs the programs and checks if the conditions given in the tests are true. We created between 4 and 17 tests for each task with a median of 7. These tests checked not only if the programs were correctly fixed, but also whether any other functionality of the program was broken. The number of test cases per task differs depending on attributes such as the number of sprites.

A submitted program was considered correct if (1) the broken functionality was fixed and (2) no other functionality was broken. For three tasks we had to supplement the tests due to limitations of Whisker: As stamps (Task 12) and the lack of delay (Tasks 7 and 14) could not be tracked yet with Whisker at the time of the experiment, we used LitterBox to analyse if the matching solution patterns were implemented and to check whether the bug pattern was removed.

During the creation of the first version of the Whisker tests (and for Tasks 7, 12 and 14 the LitterBox analyses), we checked with individual programs if the automated results are correct. Then, we rated 420 random programs (30 per task) manually and used the 13 deviations to refine the Whisker tests. After this refinement step we rated another subset of 210 randomly chosen programs (15 per task) manually: This procedure confirms a very good inter-rater reliability ($K = .96$) and thus reliable automated results.

3.3.2 Hint Evaluation. After submitting their solution for a task participants of Group Trmt evaluated the hint on a 5-point Likert scale regarding its support with debugging. Participants were then also asked to explain their rating. This provided us with qualitative data, and we analysed the comments using thematic analysis [2]: Themes are first collected, then counted and in a final step again related to the original data and our research questions. To ensure inter-rater reliability two raters (one author and one assistant) classified a randomly chosen subcorpus of 35 statements (five per Task 1–7) and agreed on a coding scheme. Then, each rater rated half of the statements and additionally five random statements per task. The comparison of these 35 statements confirms a strong inter-rater agreement with $K = .96$ and thus reliable automated results.

3.3.3 Efficiency. We used a Wilcoxon rank sum test to measure if the time differs significantly between Group Trmt and Group Ctrl and used the Vargha and Delaney $A_{12}$ measure to calculate the effect size. If $A_{12} = .50$, there are no effects in favour of any group. Values below .50 indicate effects in favour of Group Ctrl and values over .50 indicate effects in favour of Group Trmt. The time for completing a task was determined using the website on which we provided the tasks. For each task, we started measuring the time when a participant accessed the page of the respective task (where the task is described and the participants download and upload the program). Time measurement for the task was stopped when the participants submitted their final program. Time spent for giving feedback to the fictitious student, and for Group Trmt rating the hint, were not included in the time we tracked.

4 RESULTS

4.1 RQ1: Effects of Showing Hints

To answer RQ1 we consider Tasks 1 to 7, where participants of Group Ctrl had to work without hints and participants of Group Trmt were shown hints. Figure 8 and Table 7 deal with the hint evaluation: Figure 8 shows how the participants rated the hint on a five-point Likert scale. Participants were asked to explain their rating and Table 7 shows how often different aspects of the hints were mentioned.

4.1.1 Effectiveness: Differences in Terms of Functionality. Figure 4 shows how many correct solutions were produced per task and

Figure 4: Proportion of correct solutions for Tasks 1 to 7.

Table 3: Effect sizes for the functionality of Tasks 1 to 7. Effect sizes associated with significant $p$-values ($p < .05$) are bold.

| Task 1 | Task 2 | Task 3 | Task 4 | Task 5 | Task 6 | Task 7 |
|--------|--------|--------|--------|--------|--------|--------|
| OR     | OR     | OR     | OR     | OR     | OR     | OR     |
| All    | 2.20   | 11.73  | 2.26   | 44.38  | .49    | 8.31   | 22.00  |
| Experienced | 2.35  | 16.45  | 1.75   | 91.88  | .21    | 13.78  | 20.04  |
| Inexperienced | 1.89  | 8.98   | 3.82   | 27.50  | 1.24   | 4.67   | 37.50  |

3.4 Threats to Validity

Threats to external validity result from our choice of participants, programs and bugs: Our participants consisted only of primary school teachers in training, so effects for teachers at other levels might be different. However, we believe that primary school teachers are one of the most important target groups. We selected a subset of bug patterns and other bug patterns might have other particularities. As we used common bug patterns, many instances of bugs are covered and as we used seven of them, the general insights might be applicable to other bug patterns. Furthermore, our created or adapted programs differ in their complexity. This is why we included the complexity of the programs as a possible explanation of certain results. Threats to internal validity might result from our choice of participants, programs and bugs: Our participants consisted only of primary school teachers in training, so effects for teachers at other levels might be different. However, we believe that primary school teachers are one of the most important target groups. We selected a subset of bug patterns and other bug patterns might have other particularities. As we used common bug patterns, many instances of bugs are covered and as we used seven of them, the general insights might be applicable to other bug patterns. Furthermore, our created or adapted programs differ in their complexity. This is why we included the complexity of the programs as a possible explanation of certain results. Threats to construct validity arise since we cannot easily measure whether participants have, or can detect, misconceptions. However, we combined several measurements: In this paper, we will evaluate the functionality and time needed. As a next step we will evaluate the feedback the participants gave to the fictitious students.
4.2.1 Effectiveness: Differences in Terms of Functionality

Looking at individual tasks, results are similar to those regarding functionality. The hint on task 5 again significantly improves the performance, but again not significantly (p=.013, OR = .64). A slight edge over Group Trmt produced significantly more correct programs (p=.013, OR = .60). For Task 7, only the experienced participants of Group Trmt submitted significantly more correct solutions (p=.013, OR = .59).

4.2.2 Effects of Knowing Patterns

Table 6: Proportion of correct solutions for Tasks 8 to 14.

[Table showing the proportion of correct solutions for each task for inexperienced and experienced participants.]

Summary (RQ1) hints on patterns improve the performance of debugging and fixing programs. In general, programs can be fixed significantly more often and faster. In our experiment, one hint significantly improved the debugging performance.

Figure 5: Time needed to submit a program for Tasks 1 to 7.

[Figure showing the time needed to submit a program for each task for inexperienced and experienced participants.]

Effects of Hints on Debugging Scottish Programs

Figure 4: Time needed for Tasks 1 to 7.

[Figure showing the time needed for each task for inexperienced and experienced participants.]
5 DISCUSSION

5.1 General Insights

5.1.1 RQ 1: Showing Hints. The results on RQ 1 show that generic hints on bug patterns on average lead to debugging programs more often and faster. The increased effectiveness is important as teachers can only help the student to debug its program when understanding how to deal with the bug themselves. The increased efficiency implies that primary school teachers can react earlier to their student’s problem—either by telling that they are not able to fix the bug or by trying to help the student to fix the bug. This is important as one teacher has to deal with many different problem-solving approaches at the same time [31]. When hints are received it might be easier to deal with this challenge as less time is needed per bug. Still, when encountering hints for the first time, participants have to read, understand, and implement these hints, which also takes time. It needs further investigation to find out if seeing the same hints repeatedly in different programs, such that re-reading and comprehending does not need to be repeated, further reduces the amount of time needed.

We observed that more experienced participants benefit even slightly more from showing hints than inexperienced participants. Only for the Tasks 3 and 5 experienced participants benefit less in terms of effectiveness. The complexity of the program for Task 3 is very low (Table 2) and experienced participants might not need the hint to locate the bug. For Task 5, the task complexity does not stand out. However, its hint might harm the performance of experienced participants as they would have actually known where to look for the bug but were misled by the hint. However, for the five other tasks, experienced participants benefit more. This might be because basic knowledge is needed to understand the hints. Therefore, tool-support is especially useful for trained teachers in practice to face the challenge of multiple bugs at the same time.

5.1.2 RQ 2: Knowing Patterns. The results of RQ 2 do not indicate that learning opportunities with or without hints can be transferred significantly better. It seems that having repaired the corresponding bug pattern once does not suffice to transfer the hint’s explanation. Still, it is an open question if repairing the same bug patterns more times would change the findings regarding the performance with vs. without hints. Another consideration is that the explanation is only displayed in the hint. Users could elaborate more on the hint if they additionally had to, e.g., describe it in their own words as suggested by Marwan et al. [16]. This might enhance the effects of hints on known patterns.

Our results suggest that inexperienced participants benefit slightly more from knowing patterns by the hints—apart from Task 10 and 11. This might be related to the relatively high complexity of these two programs (Table 2), possibly overstraining inexperienced participants of both Group Ctrl and Group Trmt. This suggests that knowledge of bug patterns can only support repairing programs correctly and save time when the programming experience suffices to understand a program of a given complexity level. Indeed, for all other tasks inexperienced participants benefit more. This might be because they need more direct instruction (e.g., provided by hints) to transfer their gained experiences than experienced participants. Consequently, bug patterns might be a useful concept to include in the curriculum for teachers in training.

5.2 Insights on Hint Generation

5.2.1 Location of the Bug. For both Tasks 5 and 12 the hint impairs effective and efficient debugging. They deal with the bug pattern Message Never Received which has a peculiarity: While the hint is attached to the block sending a message, the fix requires modification of a different script which is often located in a different sprite, where that message should be handled. This is because static analysis only detects the pattern (e.g., that a message is sent but nowhere received) but it does not analyse content information (and thus does not return where the message should be received). This seems to provide too little help which matches the relatively bad rating of this hint (Fig. 8): The participants perceive the hint as misleading and cognitively activating (Table 7) which can also take time: “You
The hint supported finding and fixing the described bug.

| Task | Fully agree | Partially agree | Neither agree nor disagree |
|------|-------------|-----------------|----------------------------|
| Task 1 | 8% | 17% | 75% |
| Task 2 | 6% | 6% | 88% |
| Task 3 | 11% | 10% | 80% |
| Task 4 | 11% | 15% | 74% |
| Task 5 | 19% | 18% | 63% |
| Task 6 | 6% | 11% | 83% |
| Task 7 | 11% | 14% | 75% |

Figure 8: Rating of the hints.

Table 7: Classification of the comments on the hints.

|          | T1  | T2  | T3  | T4  | T5  | T6  | T7  |
|----------|-----|-----|-----|-----|-----|-----|-----|
| Positive |     |     |     |     |     |     |     |
| Problem solving | 17 | 34 | 38 | 29 | 21 | 33 | 26 |
| Clear hint | 8 | 14 | 13 | 16 | 16 | 19 | 20 |
| Localisation | 20 | 14 | 12 | 9  | 10 | 11 | 5  |
| Single components | 17 | 25 | 8  | 14 | 5  | 7  | 2  |
| Attention drawing | 12 | 13 | 7  | 7  | 8  | 9  | 7  |
| General assistance | 7 | 6 | 5 | 7 | 9 | 4 | 6 |
| Importance | 6 | 7 | 7 | 6 | 0 | 1 | 10 |
| Formulating feedback | 0 | 2 | 1 | 2 | 1 | 2 | 1 |
| TOTAL | 87 | 115 | 91 | 90 | 70 | 86 | 77 |

|          | T1  | T2  | T3  | T4  | T5  | T6  | T7  |
|----------|-----|-----|-----|-----|-----|-----|-----|
| Neutral  |     |     |     |     |     |     |     |
| Cognitive activation | 3 | 4 | 2 | 5 | 11 | 3 | 5 |
| Partial usefulness | 0 | 1 | 1 | 2 | 0 | 0 | 1 |
| TOTAL | 3 | 5 | 3 | 7 | 11 | 3 | 6 |

|          | T1  | T2  | T3  | T4  | T5  | T6  | T7  |
|----------|-----|-----|-----|-----|-----|-----|-----|
| Negative |     |     |     |     |     |     |     |
| Misleading | 3 | 1 | 0 | 6 | 24 | 1 | 3 |
| Comprehension problems | 3 | 2 | 7 | 6 | 8 | 2 | 7 |
| Insufficient assistance | 5 | 1 | 3 | 3 | 3 | 1 | 6 |
| No problem solving | 0 | 1 | 3 | 3 | 3 | 1 | 7 |
| Time | 2 | 0 | 5 | 2 | 1 | 0 | 1 |
| No deeper understanding | 0 | 3 | 0 | 1 | 0 | 2 | 3 |
| TOTAL | 13 | 8 | 18 | 21 | 39 | 7 | 27 |

|          | T1  | T2  | T3  | T4  | T5  | T6  | T7  |
|----------|-----|-----|-----|-----|-----|-----|-----|
| Unnecessary |     |     |     |     |     |     |     |
| Generally unnecessary | 3 | 3 | 4 | 3 | 4 | 1 | 1 |
| Independent solving | 3 | 1 | 1 | 0 | 4 | 2 | 1 |
| Alternative solution | 0 | 0 | 3 | 0 | 0 | 1 | 0 |
| TOTAL | 6 | 4 | 8 | 3 | 8 | 4 | 2 |

Participants highlighted the support with problem solving (Table 7). This might be explained by an obvious difference to other hints that misled some participants: The hint explains two possible solutions: (1) A message that is already sent in another script has to be selected or (2) a new message has to be sent in another script. Again, the automatic static analysis tools cannot detect whether an appropriate message is available (solution 1) or not (solution 2). The only other hint that explains two possible solutions is the hint on the bug pattern Missing Clone Initialisation. However, the two provided solutions only differ in the block that is inserted and may therefore not be crucial. Consequently, attention must be paid if the results of the static analysis lead to several quite different solutions: Hints should then either provide information about which solution should be used in which context or at least mention that all solutions should be tried out until the intended output is reached.

5.2.3 Degree of Detail. We observed the highest effects regarding the effectiveness of hints not only in Task 7 but also in the corresponding transfer Task 14. This is interesting, as the used bug pattern Stuttering Movement certainly is more open to automatic refactoring than all other bug patterns of this study. Consequently the LitterBox hint presents itself quite customised and describes very detailed what should be done to fix the bug (Table 1). The classification of hint comments confirms that the hint was perceived as clear, important and least unnecessary (Table 7). Maybe other bug patterns might not be so easily spotted, grasped and fixed by learners per se, due to greater variety of bug pattern instances or/and greater variety of appropriate fixing pathways and fixing outcomes. They consequently might need far more attention during learning than just trying to solve an individual task. Compared to the effectiveness, however, the hint on the bug pattern Stuttering Movement is slightly less helpful in terms of efficiency. This might be because five blocks have to be changed—instead of one for the other bug patterns. Thus, reading or remembering the relatively long hint might take a while. In conclusion, hints need to provide clear information about how to remove the bug pattern.

6 CONCLUSIONS

Teachers are faced with bugs and have to be able to identify and repair them. In this paper, we empirically evaluated to what extent hints can support primary school teachers in training. Our experiment confirmed that hints can help repairing bugs in students’...
ACKNOWLEDGMENTS

This work is supported by the Federal Ministry of Education and Research through project “primary::programming” (01JA2021) as part of the “Qualitätsoffensive Lehrerbildung”, a joint initiative of the Federal Government and the Länder. The authors are responsible for the content of this publication.

REFERENCES

[1] Vincent Aleven, Ido Roll, Bruce M McLaren, and Kenneth R Koedinger. 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. International Journal of Artificial Intelligence in Education 26, 1 (2016), 205–223.

[2] Manfred Max Bergman. 2010. Hermeneutic content analysis: Textual and audio-visual analyses within a mixed methods framework. SAGE Handbook of Mixed Methods in Social and Behavioral Research (2nd ed). Thousand Oaks, CA: SAGE (2010), 379–396.

[3] Bryce Bose, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of scratch projects. SIGCSE 2013 – Proceedings of the 44th ACM Technical Symposium on Computer Science Education, 215–220. https://doi.org/10.1145/2445196.2445265

[4] G. Fraser, U. Heuer, N. Körber, F. Obermüller, and E. Wasmeier (accepted). 2021. LitterBox: A Linter for Scratch Programs. In 2021 IEEE/ACM 49th International Conference on Software Engineering (ICSE).

[5] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In ITiCSE ’20: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. 89–95. https://doi.org/10.1145/3341525.3387389

[6] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. In Proceedings of the 2018 ACM Conference on International Computing Education Research (Espoo, Finland) (ICER ’18). Association for Computing Machinery, New York, NY, USA, 160–168. https://doi.org/10.1145/3230977.3231002

[7] Maurice H Halstead. 1977. Elements of software science. (1977).

[8] Stuart Hansen and Erica Eddy. 2007. Engagement and Frustration in Programming Languages and Human-Centric Computing (VL/HCC). 337–338. https://doi.org/10.1145/1227310.1227407

[9] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programmers? A controlled experiment on Scratch programs. In Proceedings of the 38th SGCC Technical Symposium on Computer Science Education (SGCSE ’07). ACM, New York, NY, USA, 271–275. https://doi.org/10.1145/1227310.1227407

[10] Maurice H Halstead. 1977. Elements of software science. (1977).

[11] Manu Kapur and Katerine Bielaczyc. 2012. Designing for productive failure. Journal of the Learning Sciences 21, 1 (2012), 45–83.

[12] Cazembe Kennedy, Aubrey Lawson, Yvon Feaster, and Eileen Kraemer. 2020. Misconception-Based Peer Feedback: A Pedagogical Technique for Reducing Misconceptions. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (Trondheim, Norway) (ITiCSE ’20). Association for Computing Machinery, New York, NY, USA, 391–397. https://doi.org/10.1145/3341525.3387411

[13] John Maloney, Mitchell Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. ACM Transactions on Computing Education (TOCE) 10 (1 2010), 16. https://doi.org/10.1145/1868358.1868363

[14] John Maloney, Mitchell Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. ACM Transactions on Computing Education (TOCE) 10 (1 2010), 16. https://doi.org/10.1145/1868358.1868363

[15] Samih Marwan, Anay Dombe, and Thomas W. Price. 2020. Unproductive Help-Seeking in Programming: What It Is and How to Address It. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (Trondheim, Norway) (ITiCSE ’20). Association for Computing Machinery, New York, NY, USA, 54–60. https://doi.org/10.1145/3341525.3387394

[16] Samih Marwan, Joseph Jay Williams, and Thomas Price. 2019. An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In Proceedings of the 2019 ACM Conference on International Computing Education Research. 61–70.

[17] Thomas Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In Proceedings of the 10th International Conference on Educational Data Mining. 192–197.

[18] Thomas W. Price, Yihan Dong, and Tiffany Barnes. 2016. Generating Data-driven Coder-centric Hints for Open-ended Programming. In EDM. 191–198.

[19] Thomas W. Price, Zhongxiao Liu, Veronica Caté, and Tiffany Barnes. 2017. Factors Influencing Students’ Help-Seeking Behavior While Programming with Human and Computer Tutors. In Proceedings of the 2017 ACM Conference on International Computing Education Research (Tacoma, Washington, USA) (ICER ’17). Association for Computing Machinery, New York, NY, USA, 127–135. https://doi.org/10.1145/3105726.3106179

[20] Kelly Rivers and Kenneth Koedinger. 2015. Data-Driven Hint Generation in Visual Solution Spaces: A Self-Improving Python Programming Tutor. International Journal of Artificial Intelligence in Education 27 (10 2015). https://doi.org/10.1007/s10799-015-9179-0

[21] Sue Sentance and Andrew Cozmadia. 2017. Computing in the curriculum: Challenges and strategies from a teacher’s perspective. Education and Information Technologies 22, 1 (2017), 469–495.

[22] Rashidb Singh, Sumit Gulwani, and Armando Solar-Lezama. 2012. Automated Feedback Generation for Correcting Introductory Programming Assignments. ACM SIGPLAN Notices 48 (04 2012). https://doi.org/10.1145/2491956.2462195

[23] Julia Soria. 2018. Misconceptions and the Beginner Programmer. 171–187.

[24] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically: In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 165–175.

[25] Alaaeddin Swidan, Felurane Hermans, and Marilee Smit. 2018. Programming Misconceptions for School Students. In Proceedings of the 2018 ACM Conference on International Computing Education Research (Espoo, Finland) (ICER ’18). Association for Computing Machinery, New York, NY, USA, 151–159. https://doi.org/10.1145/3230977.3230995

[26] Peeratham Techapalok and Elb Tilevich. 2017. Quality sounds – An online code smell analyzer for scratch programs. In 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 337–338. https://doi.org/10.1109/VLHCC.2017.8103498

[27] Wengren Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas W. Price. 2020. Step Tutor: Supporting Students through Step-by-Step Example-based Feedback. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (Trondheim, Norway) (ITiCSE ’20). Association for Computing Machinery, New York, NY, USA, 391–397. https://doi.org/10.1145/3341525.3387411

[28] Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2016. Expanding Program Elements to Novices (N). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 283–288.