Exploring Programming Task Creation of Primary School Teachers in Training

Luisa Greifenstein
luisa.greifenstein@uni-passau.de
University of Passau
Passau, Germany

Ute Heuer
ute.heuer@uni-passau.de
University of Passau
Passau, Germany

Gordon Fraser
gordon.fraser@uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

Introducing computational thinking in primary school curricula implies that teachers have to prepare appropriate lesson material. Typically this includes creating programming tasks, which may overwhelm primary school teachers with lacking programming subject knowledge. Inadequate resulting example code may negatively affect learning, and students might adopt bad programming habits or misconceptions. To avoid this problem, automated program analysis tools have the potential to help scaffolding task creation processes. For example, static program analysis tools can automatically detect both good and bad code patterns, and provide hints on improving the code. To explore how teachers generally proceed when creating programming tasks, whether tool support can help, and how it is perceived by teachers, we performed a pre-study with 26 and a main study with 59 teachers in training and the LITTERBOX static analysis tool for SCRATCH. We find that teachers in training (1) often start with brainstorming thematic ideas rather than setting learning objectives, (2) write code before the task text, (3) give more hints in their task texts and create fewer bugs when supported by LITTERBOX, and (4) mention both positive aspects of the tool and suggestions for improvement. These findings provide an improved understanding of how to inform teacher training with respect to support needed by teachers when creating programming tasks.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; **K-12 education**; • **Software and its engineering** → **Visual languages**.

KEYWORDS

assignments, automated feedback, block-based programming, elementary school, LitterBox, preservice teacher education, Scratch

Figure 1: Hint on the the bug pattern *Missing Loop* of a teacher's SCRATCH program provided by LITTERBOX.

1 INTRODUCTION

Programming tasks can be used to foster computational thinking [25] and represent a crucial part of computer science lessons [37]. Such tasks usually show example code to young learners before they create their own code, such as in the Use-Modify-Create framework [23], the PRIMM approach [41], or the TIPP&SEE strategy [39]. When this example code, however, is of poor quality, learners might adopt bad programming habits and their learning might be impeded [18]. This problem is further exacerbated at primary school level: Computer science related topics have often only recently been introduced to the curricula [17, 32] and such changes often go along with various issues [22, 38, 49]. Furthermore, particularly primary school teachers consider their lacking subject knowledge a challenge [13, 40]. As a consequence, primary school teachers may struggle when creating example code for their programming tasks.

Professional programmers receive feedback on their code from automated analysis tools. In principle, such tools may also provide scaffolding for teachers, in particular since new tools have also started to emerge for educational programming languages such as SCRATCH [24]. For example, LITTERBOX provides hints on how to improve code that contains smells or bugs [6, 15]. Figure 1 shows a bug in a teacher's example code which would certainly lead to confusion with learners, and the feedback LITTERBOX provides.

In this paper, we aim to investigate whether and how the use of a tool such as LITTERBOX also affects the creation of programming tasks. We therefore conducted an A/B study with 85 teachers in training. All participants were instructed to create a programming task for primary school children and completed a survey afterwards, and half the participants used the LITTERBOX tool during the creation. By comparing the resulting tasks and survey responses we aim to empirically answer the following research questions:

RQ 1: How do teachers in training create programming tasks?
RQ 2: How does tool support affect the task text and the code?
RQ 3: How do teachers in training perceive the tool support?

We find that (1) teachers tend to start with programming before writing a task text; (2) tool support has positive effects on code as well as task texts; and (3) teacher feedback helps us provide criteria for effective (tool) support. Consequently, we derive recommendations to inform teacher training on task creation.

2 BACKGROUND

2.1 Pedagogical Programming Tasks

Exploration of how teachers create tasks should consider criteria for pedagogical tasks. Generally, pedagogical tasks are characterised by the school setting in contrast to real-world or target tasks [33]. However, students should be able to transfer their gained competencies which is why pedagogical tasks should also relate to real-world activities. Moreover, pedagogical tasks should involve the learner and have clear task specifications [11, 33]. Creating such pedagogical tasks can be seen as one essential part of designing lessons besides organising social structures and designing supportive environments [11]. Tasks also have a crucial role in the computer science classroom, e.g., in the form of exercises or examples [37] which is also reflected in courses for primary school teachers [9]. However, to our knowledge, there has not yet been a systematic evaluation of the creation of pedagogical programming tasks.

As computational thinking can be fostered by programming activities [25], programming tasks are commonly created by computer science teachers [37]. Independently of their classification (e.g., their inherent activities [5], or the representation of the problem and the solution [37]), code is often included to be debugged, completed or tested [5, 37]. This goes along with several teaching approaches that start with or even focus on tasks with given code such as the Use-Modify-Create framework: In the first two of the three phases learners are tasked to deal with given code [23]. The use phase is further structured within the PRIMM approach, where young learners work intensively with the given code as they predict, run and investigate it [41]. The learning strategy TIPP&SEE provides meta-cognitive scaffolding between the use and modify step and has been designed in particular to support diverse learners at primary school level [39]. Other approaches for primary school computer science education such as the Universal Design for Learning also consider splitting tasks into minor parts [20].

2.2 Good Example Code

The existence of tasks including example code as part of the learning process [23, 39, 41] makes it crucial to consider criteria for good code, such as being dedicated to only one (new) element [21], being kept simple [16], and adhering to criteria related to names, expressions, or decomposition [43]. When learners are exposed to example code, they might adopt good programming habits from code examples if these are well written [36]. However, when the code quality is low, learners might not only imitate this coding style: Hermans and Aivaloglou found that code comprehension decreases when learners aged 12 to 14 years are given programs that contain code smells [18]. This is problematic since understanding code is a prerequisite to, e.g., deliberate on a bug and fix it during debugging activities [36]. Such effects might even increase for primary school children [8]. There exist many suggestions on how to achieve high code quality. Therefore, Stegeman et al. analysed handbooks and interviewed instructors to develop an assessment model [43]: They derived nine main criteria, such as names, expressions or decomposition. Consequently, educators should consider code quality when designing example code, but this might be particularly challenging for primary school teachers who often consider their lacking subject knowledge a problem [13, 40].

2.3 Tool Support for Scratch Programs

Code quality can be analysed automatically by tools [4, 6, 30, 35, 44], which differ regarding the given feedback [31]: Dr. Scratch [30] and LitterBox [6], which are both accessible via a convenient web frontend, both give positive feedback (in terms of scores by Dr. Scratch and code perfumes [34] by LitterBox) and feedback on concepts. While Dr. Scratch returns general information on the detected computational thinking concepts, LitterBox returns specific hints on how to proceed regarding detected smells and bug patterns. Therefore, Dr. Scratch might be particularly effective for extending programs according to computational thinking concepts [46] and LitterBox for repairing and improving programs [15]. LitterBox thus might support creating good example code, which is why in this paper we focus on this tool.

LitterBox [6] detects bad (bugs and smells) and good code patterns (perfumes [34]). When checking programs on the web interface, hints on the detected patterns are given. As shown in Fig. 1, a typical hint consists of an explanation of the underlying concept and (for bad code patterns) a suggested improvement [6]. While the automated hints have been demonstrated to be helpful for debugging foreign code [15], it has not been investigated yet whether they support teachers in creating programming tasks.

3 METHOD

3.1 Pre-Study

We performed a preliminary study to optimise the setting of our main study and retrieve additional feedback on the tool. The pre-study was performed as part of an earlier implementation of the seminar "Computational Thinking in Primary School" at the University of Passau in Germany with 12 teachers in training and additionally in a standalone workshop with 14 teachers in training. All participants were instructed to use LitterBox for creating a task and therefore received a short instruction on the usage of LitterBox (but none regarding code quality). In contrast to the main study, participants were completely free in how often to use the tool, they had less time (about 40 instead of 60-70 minutes) and were allowed to work in groups.

The pre-study informed the design of the main study (e.g., regarding time and tool usage) and some updates of the LitterBox tool. The updates enable us to compare the evaluations of earlier versions of the tool in the pre-study with more current versions in the main study in RQ 3 (Table 1).

3.2 Participants

In the main study, 59 teachers in training (46 females and 13 males) participated. All of them are teachers in training at the University of Passau in Germany: 86.4 % are pursuing a degree in primary school education, 6.8 % in secondary school education and 8.5 % additionally or exclusively in (media) pedagogy. Within their studies, they chose the seminar "Computational Thinking in Primary School". The teachers in training were divided into two groups (Ctrl, Trmt) by their last names. This method was chosen due to the learning platform used. We ensured that the groups are balanced which resulted in 29 participants in group Ctrl and 30 participants in group Trmt and similar programming experience (Table 2). While

Table 1: Data used to answer the RQs.

RQ	Study	Group	Qualitative Data	Quantitative Data
1	main study	Trmt, Ctrl	• textual answers on the procedure ("*How did you proceed?*") and specific issues ("*Where did you have difficulties and what worked well?*")	• numerical answers on time ("*How much time did you invest in your own task?*")
2	main study	Trmt, Ctrl	• task text (task title, sub-tasks, task type [37])	• code metrics and code patterns
3	main study, pre-study	Trmt	• textual answers on positive tool evaluation ("*Note positive aspects, purposes, advantages, especially helpful hints etc.*") and negative ("*Note negative aspects, limitations, suggestions for improvement, less helpful hints etc.*")	• evaluation of the tool's usefulness on a 5-point Likert scale from "*completely true*" to "*not at all true*" ("*LitterBox can help teachers to create good example programs.*")

Table 2: Participants' prior experience with programming.

Group	School	University	Seminar Units 1–4
Ctrl	62.1 %	79.3 %	100 %
Trmt	53.3 %	80.0 %	100 %

group Ctrl did not receive any additional support, group Trmt were instructed to use the LITTERBOX tool at least two times. As the teachers in training had not used the tool previously, group Trmt also received a short video about its usage and how it works, as we knew from our pre-study and a related study with high school students [27] that explaining these aspects would avoid confusion.

3.3 Data Collection

Table 1 gives an overview of the collected data regarding each research question. All data were collected in the fifth unit of the seminar, in which the theory of scaffolding, the Use-Modify-Create framework [23] and different task types [10, 37] were also introduced. The teachers in training were then tasked to create their own SCRATCH programming task (consisting of a task text, a starter program, and where applicable a sample solution) for the target group of primary school children. While there were no specifications regarding, e.g., program size or task type, they were instructed to spend at least 60 to 70 minutes to create their task. The teachers in training were also advised to use a template for their task text and to design an appealing task. They were not instructed to use specific programming concepts, but were reminded to bear their self-selected learning objectives such as a specific pattern or control structure in mind. After completing their task, they submitted the task text and a starter SCRATCH program. When the chosen task type required changing or extending the code, a solution SCRATCH program was also submitted. Finally, they answered a survey consisting of four open and two closed questions for group Trmt respectively two open and one closed question for group Ctrl (Table 1). The anonymised usage of the collected data has been granted by all teachers in training.

3.4 Data Analysis

3.4.1 Qualitative analysis. We applied thematic analysis [3] on the answers to the open questions of the survey (Table 1). One author and one assistant read all statements and agreed on a coding scheme. Then one author rated all data and the assistant rated 20 % of all data independently to guarantee inter-rater reliability ($K = 0.76$).

To recognise more generalisable tendencies for our target group of primary school teachers (in training), we counted occurrences for each (sub-)category, which provides further quantitative data.

3.4.2 Quantitative analysis. We used a Wilcoxon rank sum test to measure statistical differences between groups Ctrl and Trmt with $\alpha = 0.05$, and the Vargha-Delaney $\hat{A}_{12}$ effect size, which ranges from 0 to 1. If $\hat{A}_{12} = .50$, there are no effects in favour of any group, if $\hat{A}_{12} > .50$ the values of the dependent variable are higher for group Ctrl than group Trmt and vice versa if $\hat{A}_{12} < .50$.

3.4.3 RQ 1. We analysed the answers on the open questions for RQ 1 (Table 1) qualitatively and then quantitatively to discover differences between group Ctrl and group Trmt.

3.4.4 RQ 2. We analysed the data for RQ 2 (Table 1) quantitatively to discover differences between group Ctrl and group Trmt. The created tasks provided us with metrics on the code (calculated with the LITTERBOX version from 5 January 2023) and the number of types of programming tasks (deduced from [37]). As the teachers in training were free to choose their task type, we analysed either the solution SCRATCH program or the starter SCRATCH program depending on the chosen task type: For USE tasks, there is no solution program and for MODIFY tasks, the starter programs might contain intended bugs or incomplete code patterns on purpose which would distort the analysis. This resulted in 56 analysed programs (53 solution programs and 3 starter programs).

3.4.5 RQ 3. We analysed the answers on the open questions for RQ 3 (Table 1) qualitatively and then quantitatively to discover differences between the pre- and the main study.

3.5 Threats to Validity

3.5.1 External validity. The teachers in training chose the seminar voluntarily for their studies. This implies that the cohort (1) is self-selected and (2) results might be different for less interested teachers in training on the one hand and more experienced teachers or also larger programs on the other. While participants were instructed to create their own SCRATCH programs, they were told to use a template for their task text that also contained an example task. The example task was an extend task type with two subtasks and one hint for each subtask and was aimed at collision detection. While the example was perceived as helpful, it probably led to teachers in training creating similar tasks, which is why the resulting task texts may not be generalisable to other contexts.

Table 3: Procedure of teachers in training to create a task.

(Sub-)Category	% teachers in training	
Starting point	57.7 %	
brainstorming or idea search	50.0 %	
setting of learning objectives	9.6 %	
Inspiration	30.8 %	
example task	13.5 %	
seminar material	13.5 %	
Scratch environment	11.5 %	
Sprite selection	17.3 %	
Order	71.2 %	
start with program, then task text	40.4 %	
iterative approach or trial and error	25.0 %	
start with task text, then program	15.4 %	
Insecurity or difficulties	7.7 %	

3.5.2 Internal validity. The study started with a short explanation that students can learn from the example code in the tasks which is why the code should be correct, readable and not promote misconceptions. However, this probably increased the attention to code quality thus subsuming some of the impact of the tool support. While group Trmt were instructed to use the LitterBox tool at least twice, we did not check their behaviour. Moreover, they were not obliged to implement the suggestions of LitterBox. A more formal setting or including how much they actually used the output of a tool could mitigate this threat.

4 RESULTS

To answer the research questions posed in Section 1, we consider the task texts, Scratch programs and survey results as described in Section 3.

4.1 RQ 1: Procedure

On average, the teachers in training spent 82.52 minutes to create their task. Table 3 shows the (sub-)categories identified in the responses about how they proceeded when creating their tasks.

4.1.1 Starting point. Before starting the actual task creation, teachers in training tend to brainstorm ideas (Table 3). The importance of setting learning objectives, however, seems to be underestimated by teachers in training (Table 3). Maybe it is (1) considered an obvious part of creating a task that it does not have to be mentioned; (2) done very quickly and considered a minor and thus not noteworthy part of creating a task; or (3) forgotten about and thus not considered in the answer. The latter two explanations would be problematic as setting goals and related criteria is a crucial aspect when carefully designing tasks [7], for example for assessing learning outcomes.

4.1.2 Inspiration. The teachers in training often required some kind of inspiration when brainstorming ideas (Table 3): "*I looked at all examples again that we have encountered so far. It was not easy for me to come up with my own idea.*" (Trmt, P66) Besides the example task or other seminar material, some teachers in training took inspiration from Scratch sprites, backgrounds, and tutorials.

4.1.3 Sprite selection. Sprites are not only a source of inspiration, but their selection can be seen as an explicit step of task creation (Table 3), which shows the non-negligible role of figures in Scratch.

4.1.4 Order. There are two alternatives of where to start creating the task content: with programming, or with writing the task text. Most teachers in training start—and partially also end—with the code, for example one participant "*created the initial scenario in Scratch; wrote the task description; created the solution*" (Ctrl, P46). Additionally, trial and error, iterative, and even parallel approaches are partially used (Table 3) like "*at the same time, I thought about possible task texts and noted hints for the solution*" (Trmt, P16).

4.1.5 Insecurity. Some teachers in training perceived the creation of tasks as difficult (Table 3), in particular because it was their first attempt to design a complete task on their own. This matches the finding that especially primary school teachers consider their lacking knowledge a challenge [40].

4.1.6 Specific issues. While we found no significant differences in the previous categories, when asked if they had specific issues, group Trmt reported significantly fewer programming difficulties ($p = 0.038$, $\hat{A}_{12} = 0.66$) and group Ctrl stated that "*creating compact code*" (Ctrl, P32) and "*programming efficiently*" (Ctrl, P33) was difficult.

> **RQ 1 Summary.** Before creating their own tasks, the teachers in training often search for ideas and take inspiration from other examples. Generally, teachers in training tend to start with programming and write the task text afterwards.

4.2 RQ 2: Effects of Tool Support on the Tasks

All tasks (both task texts and Scratch programs) of both groups are available at https://doi.org/10.6084/m9.figshare.22657402.

4.2.1 Task text. The example task consisted of two subtasks (both were 'extend' task types) and two hints. This should be kept in mind when interpreting the following results. The tasks of the teachers in training had 2.28 subtasks on average with a median of 2. Of the teachers in training, 70 % created only 'extend' tasks in all of their subtasks. Other task types [37] that were implemented in subtasks could be classified as 'create with given code', 'debug' and 'test' task types. 'Optimising' or 'explaining', e.g., were never realised even though these task types were presented in the seminar. Regarding the task titles, the topics seem to focus on animals (34.5 %), ball games (20.7 %), every day life such as food (17.2 %) and fantasy worlds (13.8 %). However, other school subjects than sports were only addressed in 3.4 % of task titles. Group Trmt gave significantly more hints in their task text than group Ctrl ($p = 0.01$, $\hat{A}_{12} = 0.33$). Group Ctrl inserted 1.28 ($\tilde{x} = 2$) and group Trmt 1.9 ($\tilde{x} = 2$) hints on average. This might relate to group Trmt experiencing the tool's hints as helpful and discovering strategies for elaborated feedback on concepts and on how to proceed. They might therefore also want to scaffold their own tasks more.

4.2.2 Code metrics and code patterns. The created Scratch programs contain 3.44 sprites ($\tilde{x} = 3$), 7.53 scripts ($\tilde{x} = 5$) and 58.42 blocks ($\tilde{x} = 40$) on average. Considering the patterns found in the code, the programs contain on average 0.98 different bug patterns ($\tilde{x} = 1$), 2.71 smells ($\tilde{x} = 2$) and 5.91 perfumes ($\tilde{x} = 6$). However, the

Table 4: Evaluation of positive aspects of the LitterBox tool for creating tasks.

(Sub-)Category	% teachers in training	
Functionality	94.1 %	
detection of patterns	70.6 %	
suggestions for improvement	54.9 %	
other	17.6 %	
Advantages for teachers	29.4 %	
time	9.8 %	
additional help	9.8 %	
affective help	5.9 %	
other	5.9 %	
Representation	27.5 %	
Usefulness for students	19.6 %	

Table 5: Evaluation of negative aspects of the LitterBox tool for creating tasks.

(Sub-)Category	% teachers in training	
Functionality	33.3 %	
no solving of all problems	11.8 %	
no analysis of semantics	9.8 %	
incorrect analysis	7.8 %	
no automatic refactoring	5.9 %	
Representation	33.3 %	
comprehension problems	25.5 %	
usability	9.8 %	
None	27.5 %	
No usefulness for students	7.8 %	

number of smells and bug patterns differs significantly between group Ctrl and group Trmt: When supported by LitterBox, teachers in training inserted significantly fewer bug patterns ($p = 0.034$, $\hat{A}_{12} = 0.66$) and smells ($p = 0.022$, $\hat{A}_{12} = 0.68$). Group Ctrl inserted 1.26 ($\tilde{x} = 1$) bug patterns while Group Trmt inserted 0.72 ($\tilde{x} = 0$) bug patterns. Group Ctrl inserted 3.19 ($\tilde{x} = 3$) smells while group Trmt inserted 2.28 ($\tilde{x} = 2$) smells. This suggests that the LitterBox tool is helpful for debugging programs and for improving the code quality. To avoid that this finding is only an artefact of smaller programs, we also calculated the bug density and smell density with $\#bugs/\#blocks$ and $\#smells/\#blocks$ which is still significantly lower for group Trmt than group Ctrl (bugs: $p = 0.034$, $\hat{A}_{12} = 0.66$; smells: $p = 0.022$, $\hat{A}_{12} = 0.68$). We also looked at the individual patterns and (despite the small sample size) found that the *Missing Loop* bug pattern occurs significantly more often in the programs of group Ctrl ($p = 0.035$, $\hat{A}_{12} = 0.57$). This could be because the bug pattern is generally very common and the corresponding LitterBox hint (Fig. 1) is easy to implement.

> **RQ 2 Summary.** LitterBox supports teachers in training with creating more scaffolded task texts and less faulty programs while there were no effects on, e.g., program size or the task type.

Figure 2: Rated usefulness of the tool for creating tasks.

4.3 RQ 3: Tool Evaluation

The categories of the textual answers provide information about positive (Table 4) and negative aspects (Table 5) of the tool support.

4.3.1 General evaluation. Overall, the teachers in training were in favour of the tool support and no teacher in training disagreed with its usefulness (Fig. 2). This aligns with primary school teachers considering tools useful for giving feedback (even though they often were not aware of tools) [13]. When directly asked about negative aspects of and suggested improvements for the tool, 27.5 % of teachers in training stated that they cannot think of any (Table 5).

4.3.2 Functionality. Teachers in training like that the LitterBox tool detects different types of patterns and gives suggestions for improvement (Table 4). Indeed, most LitterBox hints are composed of feedback on concepts and feedback on how to proceed [6, 31], "*that makes it easier for inexperienced teachers to recognise and fix bugs*" (Trmt, P82). Teachers in training perceived some missing functionality as negative (Table 5). While incorrect analyses such as false positives are a common problem of automated feedback tools and should therefore be explained beforehand to enable users to interpret automated hints [27], other criticised aspects are out of the scope of the LitterBox tool such as the analysis of semantics (Table 5): "*Litterbox cannot judge, if my code makes sense semantically*" (Trmt, P35). This is true as LitterBox is a static analysis tool that analyses the code patterns but not the output [6]. Semantics could be analysed with dynamic analysis tools such as the Whisker test framework for Scratch [42]. Another interesting suggestion of the teachers in training was to perform automatic refactoring (as for example already explored for Scratch programs [1, 45]) as "*you still have to repair the bug yourself and find a new solution*" (Trmt, P61). Automatic refactoring might save time for teachers, although one needs to be careful regarding learning processes: Elaborated feedback involves and motivates the learner (both at teacher or student level) more than a provided or automatically generated solution [14, 31].

4.3.3 Advantages for teachers. Teachers in training perceive several advantages especially for teachers such as the additional help they receive, or help in terms of affective factors (Table 4). LitterBox might moreover save time as "*bugs can be detected without having to search and try for a long time*" (Trmt, P69) and you "*get feedback in the process and are able to react to it immediately*" (Trmt, P42). This refers to some keys of effective feedback such as being timely, actionable and ongoing or formative [48]. This could support teachers regarding the frequently identified lack of time for planning and during computer science lessons [40, 50], as for example reported in a recent study on the efficiency of automated hints for debugging Scratch programs [15]. Apart from timing issues, automated tools might also provide some support regarding the reported relatively low computer science self-esteem [47] and programming

self-concept [13] of primary school teachers as "*it is not only shown what is wrong or messy, but also what you have done well. That is motivating*" (Trmt, P64) (i.e., it reports code perfumes [34]).

4.3.4 Representation. The way LITTERBOX represents its findings was mentioned both positively and negatively (Tables 4 and 5). The teachers in training are in favour of the tool's illustration such as "*the visual representation is always great*" (Trmt, P5). However, teachers in training often had comprehension problems. These are significantly lower in the main compared to the pre-study ($p = 0.048$, $\hat{A}_{12} = 0.62$). Further investigation is needed on whether this results from the revised study design or improvements of the tool.

4.3.5 (Not) for students. While the teachers in training were asked about positive and negative aspects of the LITTERBOX tool for creating tasks, some teachers in training also noted their opinion on the usefulness of LITTERBOX for students. While some teachers in training consider the tool "*functional for primary school students to check themselves*" (Trmt, P1) and to get a "*sense of achievement without the teacher's help*" (Trmt, P6), others state that "*for a primary school child, Litterbox would probably seem overwhelming, so it is only useful for the teacher*" (Trmt, P67). Therefore, more research is needed on whether LITTERBOX is usable by young learners, how they can be activated, for example using self-explanation prompts [26], and if more straightforward hints are needed.

> **RQ 3 Summary.** The teachers in training consider it helpful to get feedback on code patterns during the creation of a task but suggest some extensions such as automatic refactoring.

5 DISCUSSION

5.1 Teacher Training in General

One way for research to actually propagate to the classroom is to inform teacher training, which is a main strategy to counteract challenges such as lacking knowledge or confidence [13]. Mason and Rich [28] performed a systematic literature review of 21 studies on computing education training for primary school teachers and found that teacher training can be effective both for increasing knowledge and for improving attitudes. The knowledge teachers gain during training can be categorised into technological knowledge (TK), pedagogical knowledge (PK) and content knowledge (CK) according to the TPACK framework [19]. We consider these three categories regarding our results and related work on the support needed by teachers when creating programming tasks.

5.2 Pedagogical Knowledge of Tasks

We found that teachers in training need ideas for their tasks (RQ 1). Experienced teachers draw inspiration from existing material [13] from, e.g., Code.org (https://code.org/), Teach Computing (https://teachcomputing.org/primary-teachers), the Raspberry Pi Foundation (https://raspberrypi.org/), or publicly shared SCRATCH projects. When selecting existing or creating new programs, gender differences should also be considered, i.e., "*I thought of a topic that is appealing for both boys and girls*" (Trmt, P42), which is in-line with prior research: While both girls and boys like programs with animals, girls' programs rather involve, e.g., music and dancing and boys' programs, e.g., soccer [12]. In our study, shared preferences and

those of boys were applied, but girls' preferences are rather underrepresented. These preferences show a need for differentiation, but also allow for cross-curricular teaching with, e.g., artistic subjects, which again opens up a wide range of ideas for tasks.

5.3 Content Knowledge of Programming

We found that teachers in training often have difficulties when programming (RQs 1 and 2). Teacher training is known to be effective for promoting subject knowledge [28], for example by letting primary school teachers try out a variety of task types [9]. This might not only help with pedagogical knowledge but also with content knowledge, as the teachers are exposed to (appropriate) example code. This could also reduce affective issues, as low confidence is related to lacking subject knowledge [13].

5.4 Technological Knowledge of Tools

We found that even when primary school teachers in training struggle with programming they can be supported with tools such as LITTERBOX (RQs 1 to 3). Teachers also have difficulties supporting students during programming [29, 50], which again could be supported with automated analysis tools. However, even in-service teachers are often unaware of available tools, although they consider them useful once they are presented to them [13]. We therefore suggest that teacher training should introduce automated analysis tools. Since programming is the most common approach to foster computational thinking [2], and SCRATCH is a very popular programming environment, tools for SCRATCH could often be meaningfully integrated into teacher training.

6 CONCLUSIONS

Example code is an essential component of educational programming tasks, but inadequate example code may negatively affect learning. Unfortunately, primary school teachers often have insufficient programming knowledge to produce good example code when preparing tasks. In this paper we studied whether and how the support of a static code analysis tool, LITTERBOX, influences teachers in training when creating tasks. While the study led to important suggestions on how to improve LITTERBOX further, we also found positive effects on code as well as the task text.

While a primary aim of our study is to inform teacher training, it also has implications on future research in code analysis tools. For example, analysis tools are built to simply report all code issues they encounter. However, depending on the task type there can be different types of code examples, such as starter code, intentionally buggy code, or solution code, and each type of program may merit different types of analyses. Future research could also consider further tools and types of analysis, as well as the use of deep learning techniques to analyse not only the code in isolation, but in conjunction with the corresponding task text.

REFERENCES

[1] Felix Adler, Gordon Fraser, Eva Gründinger, Nina Körber, Simon Labrenz, Jonas Lerchenberger, Stephan Lukasczyk, and Sebastian Schweikl. 2021. Improving Readability of Scratch Programs with Search-based Refactoring. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 120–130.

[2] Maria Ausiku and Machdel Matthee. 2021. Preparing Primary School Teachers for Teaching Computational Thinking: A Systematic Review. In *International Symposium on Emerging Technologies for Education, International Conference on Web-Based Learning*. Springer, 202–213.

[3] 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. Thousand Oaks, SAGE* (2010), 379–396.

[4] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of scratch projects. *Proc. ACM Technical Symposium on Computer Science Education*, 215–220.

[5] Matt Bower. 2008. A taxonomy of task types in computing. In *Proceedings of the Conference on Innovation and Technology in Computer Science Education*. 281–285.

[6] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. Litterbox: A linter for scratch programs. In *Proc. Int. Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 183–188.

[7] Ursula Fuller, Colin G Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L Lewis, Donna McGee Thompson, Charles Riedesel, et al. 2007. Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin* 39, 4 (2007), 152–170.

[8] Alexandra Funke, Katharina Geldreich, and Peter Hubwieser. 2016. Primary school teachers' opinions about early computer science education. In *Proceedings of the Koli Calling Int. Conference on Computing Education Research*. 135–139.

[9] Katharina Geldreich, Mike Talbot, and Peter Hubwieser. 2018. Off to new shores: preparing primary school teachers for teaching algorithmics and programming. In *Proc. Workshop in Primary and Ssecondary Computing Education*. 1–6.

[10] Katharina Geldreich, Mike Talbot, and Peter Hubwieser. 2019. Aufgabe ist nicht gleich Aufgabe–Vielfältige Aufgabentypen bewusst in Scratch einsetzen. *Informatik für alle* (2019).

[11] Peter Goodyear. 2015. Teaching as design. *Herdsa review of higher education* 2, 2 (2015), 27–50.

[12] Isabella Graßl, Katharina Geldreich, and Gordon Fraser. 2021. Data-driven Analysis of Gender Differences and Similarities in Scratch Programs. In *The 16th Workshop in Primary and Secondary Computing Education*. 1–10.

[13] Luisa Greifenstein, Isabella Graßl, and Gordon Fraser. 2021. Challenging but Full of Opportunities: Teachers' Perspectives on Programming in Primary Schools. In *Koli Calling International Conference on Computing Education Research*. 1–10.

[14] Luisa Greifenstein, Isabella Graßl, Ute Heuer, and Gordon Fraser. 2022. Common Problems and Effects of Feedback on Fun When Programming Ozobots in Primary School. In *Proc. Workshop in Primary and Secondary Computing Education*. 1–10.

[15] Luisa Greifenstein, Florian Obermüller, Ewald Wasmeier, Ute Heuer, and Gordon Fraser. 2021. Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training. In *The 16th Workshop in Primary and Secondary Computing Education*. 1–10.

[16] Jean Griffin, Eliot Kaplan, and Quinn Burke. 2012. Debug'ems and other deconstruction kits for STEM learning. In *IEEE 2nd integrated STEM education conference*. IEEE, 1–4.

[17] Fredrik Heintz, Linda Mannila, and Tommy Färnqvist. 2016. A review of models for introducing computational thinking, computer science and computing in K-12 education. In *FIE '16*. 1–9.

[18] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[19] Mary C Herring, Matthew J Koehler, Punya Mishra, et al. 2016. *Handbook of technological pedagogical content knowledge (TPACK) for educators*. Vol. 3. Routledge New York.

[20] Maya Israel, Gakyung Jeong, Meg Ray, and Todd Lash. 2020. Teaching elementary computer science through universal design for learning. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 1220–1226.

[21] Takayuki Kimura. 1979. Reading before composition. *ACM SIGCSE Bulletin* 11, 1 (1979), 162–166.

[22] Laura R Larke. 2019. Agentic neglect: Teachers as gatekeepers of England's national computing curriculum. *BJET* 50, 3 (2019), 1137–1150.

[23] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *Acm Inroads* 2, 1 (2011), 32–37.

[24] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[25] Linda Mannila, Valentina Dagiene, Barbara Demo, Natasa Grgurina, Claudio Mirolo, Lennart Rolandsson, and Amber Settle. 2014. Computational thinking in K-9 education. In *ITICSE '14*. 1–29.

[26] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An evaluation of the impact of automated programming hints on performance and learning. In *Proc. ACM Conference on Int. Computing Education Research*. 61–70.

[27] Samiha Marwan, Preya Shabrina, Alex Milliken, Ian Menezes, Veronica Catete, Thomas W Price, and Tiffany Barnes. 2021. Promoting Students' Progress-Monitoring Behavior during Block-Based Programming. In *Koli Calling International Conference on Computing Education Research*. 1–10.

[28] Stacie L Mason and Peter J Rich. 2019. Preparing elementary school teachers to teach computing, coding, and computational thinking. *Contemporary Issues in Technology and Teacher Education* 19, 4 (2019), 790–824.

[29] Tilman Michaeli and Ralf Romeike. 2019. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*. 1–7.

[30] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. 2015. Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia* 46 (2015), 1–23.

[31] Susanne Narciss. 2013. Designing and evaluating tutoring feedback strategies for digital learning. *Digital Education Review* 23 (2013), 7–26.

[32] Christin Nenner and Nadine Bergner. 2022. Informatics Education in German Primary School Curricula. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer, 3–14.

[33] David Nunan. 2004. *Task-based language teaching*. Cambridge university press.

[34] Florian Obermüller, Lena Bloch, Luisa Greifenstein, Ute Heuer, and Gordon Fraser. 2021. Code Perfumes: Reporting Good Code to Encourage Learners. In *The 16th Workshop in Primary and Secondary Computing Education*. 1–10.

[35] Go Ota, Yosuke Morimoto, and Hiroshi Kato. 2016. Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 238–239.

[36] Brad Richards. 2000. Bugs as features: Teaching network protocols through debugging. In *Proc. ACM Tech. Symp. on Computer Science Education*. 256–259.

[37] Alexander Ruf, Marc Berges, and Peter Hubwieser. 2015. Classification of programming tasks according to required skills and knowledge representation. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*. Springer, 57–68.

[38] Jim Ryder. 2015. Being professional: accountability and authority in teachers' responses to science curriculum reform. *Stud Sci Educ* 51, 1 (2015), 87–120.

[39] Jean Salac, Cathy Thomas, Chloe Butler, Ashley Sanchez, and Diana Franklin. 2020. TIPP&SEE: a learning strategy to guide students through use-modify Scratch activities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 79–85.

[40] Sue Sentance and Andrew Csizmadia. 2017. Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies* 22, 2 (2017), 469–495.

[41] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teachers' experiences of using primm to teach programming in school. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 476–482.

[42] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proc. ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*. 165–175.

[43] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2014. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling international conference on computing education research*. 99–108.

[44] Peeratham Techapalokul and Eli Tilevich. 2017. Quality Hound — An online code smell analyzer for scratch programs. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 337–338.

[45] Peeratham Techapalokul and Eli Tilevich. 2019. Code quality improvement for all: Automated refactoring for Scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 117–125.

[46] Giovanni Maria Troiano, Sam Snodgrass, Erinç Argımak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. 2019. Is my game OK Dr. Scratch? Exploring programming and computational thinking development via metrics in student-designed serious games for STEM. In *Proceedings of the 18th ACM international conference on interaction design and children*. 208–219.

[47] Rebecca Vivian, Keith Quille, Monica M McGill, Katrina Falkner, Sue Sentance, Sarah Barksdale, Leonard Busuttil, Elizabeth Cole, Christine Liebe, and Francesco Maiorana. 2020. An international pilot study of k-12 teachers' computer science self-esteem. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 117–123.

[48] Grant Wiggins. 2012. Seven keys to effective feedback. *Feedback* 70, 1 (2012), 10–16.

[49] T Wolff, L Hellmig, and A Martens. 2020. STATE OF THE ART IN CURRICULUM RESEARCH FROM THE PERSPECTIVE OF GERMAN COMPUTER SCIENCE TEACHERS. *ICERI2020 Proceedings* (2020), 9177–9184.

[50] Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2016. Expanding computer science education in schools: understanding teacher experiences and challenges. *Computer Science Education* 26, 4 (2016), 235–254.