First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts

James Prather
Abilene Christian University
Abilene, Texas, USA
jrp09a@acu.edu

Raymond Pettit
Abilene Christian University
Abilene, Texas, USA
rsp05b@acu.edu

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Dastyni Loksa
University of Washington
Seattle, Washington, USA
dloksa@uw.edu

Alani Peters
Abilene Christian University
Abilene, Texas, USA
alp13d@acu.edu

Zachary Albrecht
Abilene Christian University
Abilene, Texas, USA
zca16a@acu.edu

Krista Masci
Abilene Christian University
Abilene, Texas, USA
klc10b@acu.edu

ABSTRACT

When solving programming problems, novices are often not aware of where they are in the problem-solving process. For instance, students who misinterpret the problem prompt will most likely not form a valid conceptual model of the task and fail to make progress towards a working solution. Avoiding such errors, and recovering from them once they occur, requires metacognitive skills that enable students to reflect on their problem-solving processes. For these reasons, developing metacognitive awareness is crucially important for novice students. Previous research has shown that explicitly teaching key steps of programming problem-solving, and having students reflect on where they are in the problem-solving process, can help students complete future programming assignments. Such metacognitive awareness training can be done through personal tutoring, but can be difficult to implement without a high ratio of instructors to students. We explore a more scalable approach, making use of an automated assessment tool, and conduct a controlled experiment to see whether scaffolding the problem-solving process would increase metacognitive awareness and improve student performance. We collected all code submissions by students in both control and experimental groups, as well as data from direct observation using a think-aloud protocol. We found that students who received the intervention showed a higher degree of understanding of the problem prompt and were more likely to complete the programming task successfully.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Human-centered computing** → **User studies**;

KEYWORDS

CS1, automated assessment tools, metacognitive awareness

ACM Reference Format:
James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287374

1 INTRODUCTION

Solving a programming problem requires careful navigation through multiple learning steps [25]. With only one mistake, the novice programmer could find themselves lost in the weeds without any idea as to why they are lost or stuck [36]. This is because novice programmers usually lack metacognitive awareness [14, 24, 28, 38] – the ability to think about and reflect on their problem-solving process – and cannot articulate it to a peer or instructor. The lack of this crucial skill in novice programmers could help to explain the tremendously high failure rate in computer science [4, 47], especially since the highest-performing novices tend to display some aspects of metacognitive awareness or other self-regulated learning strategies [10].

There are two general approaches to solving this problem: explicit instruction in the classroom [24, 25] or implicit training through careful design of the automated assessment tool or intelligent tutoring system used in the course, through which students complete their homework [36]. Loksa et al. [25] investigated metacognitive awareness in novice programmers using an explicit strategy by making students aware of and coaching them through six problem-solving stages: (1) *reinterpret the prompt*, (2) *search*

for analogous problems, (3) *search for solutions*, (4) *evaluate a potential solution* , (5) *implement a solution*, (6) *evaluate implemented solution*. Before receiving help, students had to identify to an instructor where they were in these six stages, therefore reflecting on their programming process and practicing metacognitive awareness. Prather et al. [36] took this framework and applied it to the design of an automated assessment tool (AAT) to understand where the tool currently failed to implicitly support novice metacognition. They found five points at which the design of the AAT adversely impacted novice metacognitive awareness and called for future work to investigate each one. Three of the difficulties they highlighted applied to the first learning stage, *reinterpret the prompt*. Their analysis revealed the importance for students to correctly interpret the problem statement from the outset. They highlighted several difficulties that resulted if an incorrect conceptual model was formed early in the problem-solving process, such as a reluctance to reconsider a misconstrued conceptual model. They hypothesize that an explicit prompt from an AAT, checking that students understand a problem initially, may be particularly useful in this regard.

Prior work has suggested that focusing on the problem prompt can provide enough scaffolding to help novices build metacognitive awareness [17]. In this paper, we investigate whether asking novice programmers to solve a randomly generated test case immediately after reading the problem prompt can help them overcome metacognitive difficulties encountered at that learning stage and, therefore, should be incorporated into the design of AATs for better implicit metacognitive training.

We investigate the following research question:

- **RQ:** Can solving a randomly generated test case immediately after reading a problem prompt help novice programmers overcome metacognitive difficulties encountered in the early stages of problem-solving?

We report on a think-aloud study with CS1 students completing a timed programming exercise in order to determine if the intervention produces a noticeable impact on metacognitive awareness in an experimental group, over those in a control group.

2 RELATED WORK

2.1 Automated Assessment Tools

As this study centers around the use of automated assessment tools (AATs) and their design, it is important to first briefly consider AATs, their history, and how they have been designed (or not) to support metacognitive awareness. There is a long history of development, testing, and improvement of AATs [1, 12, 20, 21, 26, 34]. However, these AATs are not generally designed to improve the user's metacognitive awareness [36]. The only discussion in the literature about AATs that could be related to one of the stages from Loksa's framework [25] addresses stage 5, *implement a solution*, which centers on enhancing compiler error message feedback. Compiler error messages have long been documented to be a great source of confusion and frustration to students [9, 29, 32, 46]. How novices encounter error messages has been linked to success through Jadud's EQ [22] and enhanced compiler error messages (ECEMs) have shown potential in reducing EQ and to have positive effects on similar error metrics [5, 8]. This indicates that the addition of ECEMs can help students to better understand the errors in their solution and therefore reduce the overall amount of errors encountered. Accordingly, several researchers have attempted to add ECEMs into their AATs [3, 16, 41, 48]. A few researchers have reported empirical results on the efficacy of ECEMs, though the efficacy of this approach is an ongoing debate [6, 7, 11, 35, 37]. Likewise, a few researchers have attempted to take a human-factors approach to the design of ECEMs, highlighting the importance of human-centered design in AATs [18, 29, 30, 37, 46].

2.2 Metacognition in Novice Programmers

Introductory programming courses often focus solely on syntax and data structures, but there is a growing consensus among computer science education researchers that there should be more emphasis on assisting the novice in building a mental scaffold around which they can correctly place knowledge and in doing so, develop metacognitive awareness [10, 14, 19, 25, 28, 38, 43]. Metacognitive awareness is, simply put, knowing about knowing. Applied to programming, it is not just knowledge of the problem and how to solve it, but knowledge of where one is in the problem-solving process and self-reflection on that state [31].

A few studies, such as Falkner et al. [15] and Hauswirth & Adamoli [19] report on experiments designed to measure novice programmers' metacognitive awareness, self-reflection, or self-awareness of learning strategies. The most comprehensive study to date on promoting metacognitive awareness in novice programmers is by Loksa et al. [25]. As discussed in Section 1, they identified six distinct problem-solving stages that learners iteratively progress through. They reported on an intervention at a code camp where the control group was taught how to code and the experimental group was additionally trained in these six problem-solving stages and the use of an IDE with an Idea Garden. They reported that students with this training were significantly more productive, required less help, and displayed a positive shift in growth mindset. This indicates that modified pedagogical approaches and programming environments warrant development. Some successful modifications, however, are difficult to scale or hard to implement for online learning technologies, such as massively open online courses, which Loksa et al. acknowledged as a limitation of their work.

Prather et al. [36] recently reported on an intervention designed to explore the metacognitive difficulties faced by novices when using AATs, using the work of Loksa et al. as a theoretical framework. They conducted a think-aloud study with novice programmers solving a programming problem while using Athene, an AAT with a long history of empirical work [33, 35–37, 45]. The five metacognitive difficulties they report are shown in Table 1. The first three of these identified difficulties involve forming an appropriate conceptual model of the problem. They found that the students who successfully solved the problem were consistently more likely to correctly interpret the problem statement before beginning to write code. Moreover, the most common error made by those students unable to complete the programming problem was failure to build a correct conceptual model of the problem. Prather et al. argue that the single greatest weakness of modern AATs is that they merely present a problem to be solved, assuming that students will eventually conceptualize the problem correctly and allowing the

Table 1: Observed difficulties to metacognitive awareness by novices using AATs from [36].

Metacognitive Difficulty	Explanation
Forming	Forming the wrong conceptual model about the right problem
Dislodging	Dislodging an incorrect conceptual model of the problem may not be solved by re-reading the prompt
Assumption	Forming the correct conceptual model for the wrong problem
Location	Moving too quickly through one or more stages incorrectly leading to a false sense of accomplishment and poor conception of location in the problem-solving process
Achievement	Unwillingness to abandon a wrong solution due to a false sense of being nearly done

submission of code without any assurances as to how the problem has been interpreted by the student.

3 METHODOLOGY

In this paper we investigate whether providing an explicit metacognitive prompt - one that requires solving a test case immediately after reading the problem prompt in an AAT - will assist novice programmers in overcoming the metacognitive difficulties identified by Prather et al. [36]. We presented the programming problems in our automated assessment tool, Athene, and the test cases were crafted in the Canvas LMS.

3.1 Think-Aloud Study

During the sixth week of CS1 at Abilene Christian University, one researcher, who was not the professor of record, appeared in both sections of the course to solicit student participation, discuss the study, and administer IRB requirements. Students were asked to participate in exchange for three perfect daily quiz scores, which they would receive regardless of how well they performed in the study. Not participating was guaranteed to not have any negative consequences on their grades. A total of 36 out of 41 students chose to participate. Because there were two sections of the course, the first section was chosen as the control group (n=17) and the second section as the experimental group (n=21). Students in both control and experimental groups signed up for a time slot to meet one of the researchers in the lab for an hour-long one-on-one session where the student was observed completing a programming assignment using the think-aloud protocol [39]. The session consisted of a five-minute warm-up exercise to help the student become familiar with the task of talking aloud as suggested by [44] and [49], a 35-minute programming task where the student was given a problem in Athene and asked to solve it in the allotted time, and a 20-minute detailed feedback session where each student was given personalized advice based on the observations of the researcher. This hour-long one-on-one format is based on the methodology of Prather et al. [36]. The programming language students used was C++.

The general format of the think-aloud study follows the usability testing guidelines found in Rubin and Chisnell [39], including pre- and post-testing checklists and scripts. At the beginning of each think-aloud session, the evaluator read from a script outlining the reason for the session, the goal of the session, and what was expected of the student. The warm-up exercise was to write a program that would output "Hello, world." This particular task was chosen to reduce cognitive load introduced by the programming task during the warm-up. After completing the warm-up exercise, students were asked to complete the programming task within a time limit of 35 minutes. The task was this: given n integers, compute whether there were more positive or negative integer numbers provided as input. For this problem, students would need to understand the following concepts: console input, console output, conditionals, and loops. This problem is identical to the one used by Prather et al. [36]. The course was structured in such a way that the problem correlated with course topics at the time and therefore should have been familiar but moderately challenging. While each student worked to solve the problem, a researcher took extensive notes on what the student did and said. Interactions during the programming task between the researcher and student were kept to a minimum, as per Ericsson and Simon [13]. After reading the problem prompt, the control group was allowed to immediately begin coding, while the experimental group was immediately directed to solve a test case quiz inside of Canvas simply asking students to calculate the output of the program described in the prompt given a random set of input. Once the students in the experimental group had passed the test case quiz, they were allowed to begin writing code. This meant that the experimental group had slightly less time to complete the task than the control. We discuss and reject this as a potential threat to validity in Section 3.3.

Prior work has shown that participants trained in metacognitive strategies exhibit a positive shift in growth mindset [25]. We also explored this, by investigating whether participants had a shift towards a growth mindset from pre- to post-testing scores. In both groups, before reading the problem prompt and after solving the problem or time had expired, students were asked to answer how they felt about three growth mindset statements, which we took from Scott and Ghinea [42]. One statement endorsed a fixed mindset (Q1. "I do not think I can really change my aptitude for programming") and two statements endorsed the growth mindset (Q2. "I do not have a fixed level of programming aptitude and hard work can change it"; and Q3. "I believe I am able to achieve a high level of programming aptitude, with enough practice"). Participants were asked to rate how much they agreed with the three statements on a five-point Likert scale.

3.2 Qualitative Coding Process

Participant observation allowed us to record the participants' actions, apparent thought and problem-solving process, and external reactions to error messages and other feedback. Participant-specific data were separately recorded, tagged, and then moved

into ATLAS.ti, a qualitative software analysis package. We used a collaborative coding process for the initial stage of coding the qualitative observation session notes [40]. The codebook was developed collaboratively among multiple researchers. Two researchers then independently coded 10% of the data (without knowing how the other had coded the data). A third researcher used Cohen's Kappa [2] to measure agreement between coders and found an interrater reliability rate of 67.2%, which is classified as "substantial agreement" [27]. With this initial step complete and having verified the accuracy of the double-blind coding process, a second step was taken where the two coders worked through their disagreements about which codes were used, refining the codebook [27]. Following this, the lead researcher proceeded to code the rest of the data. When the first stage was completed, we used ATLAS.ti to identify major themes within the data. These themes, grounded on the data, are explored in Section 4.2.

3.3 Threats to Validity

Four threats to the validity of this study are: 1. the potential for increased cognitive load during the think-aloud sessions; 2. the fact that the two groups were naturally split into two course sections; 3. the fact that the experimental group had slightly less time than the control group to complete the main task; and 4. students in the experimental group may have taken cues from the tasks researchers asked them to do. We address these in order here

Participants in the think-aloud study were in a one-on-one setting, were asked to think-aloud, and did not have access to any of their previously written assignments. It is possible that all of these factors increased student cognitive load in the think-aloud study. We attempted to offset this 'think-aloud factor' by adding in the warm-up exercise as suggested by Teague et al. [44] and Whalley and Kasto [49]. Furthermore, students who did not complete the warm-up exercise were given a correctly working warm-up exercise to help ensure that students were starting from a common place. In terms of writing code for the programming task, we asked all students to use Notepad, which is a plain text editor. This was to ensure that all students used the same environment, and to prevent students with knowledge of more advanced programming tools from utilizing their features for an advantage.

As the control and experimental groups were determined by the two sections of the course, we conducted a post-test analysis on midterm test scores to compare the similarity of students in each group. The mean score of students in the control group was 0.73 (SD = 0.19) and the mean score of students in the experimental group was 0.81 (SD = 0.15). An independent samples t-test (unequal variances, two-tail) showed that the difference in these scores is not statistically significant ($t(29) = 1.43$, $p = 0.16$), allowing us to conclude that that there is no significant difference in ability between the groups.

As discussed in Section 3, by design, the control group was allowed to immediately begin coding, while the experimental group was immediately directed to solve a quiz inside of Canvas consisting of a randomly generated test case. Once the students in the experimental group had passed the test case quiz, they were allowed to begin writing code. This meant that the experimental group had slightly less time to complete the task than the control. We do not believe this is a serious threat to validity for two reasons. First, no participant in the experimental group spent more than a minute on the test case quiz. Second, we believe that by requiring students to reflect on the prompt, we merely formalized a necessary step that students should have been taking anyway, and is an essential part of the study design.

Finally, the students in the experimental group may have taken their involvement in the experiment somewhat more seriously than those in the control. Students in the experimental group were asked to solve the test case before coding and so may have spent more time reading the prompt to pass the quiz and more carefully thinking about the solution where they would have otherwise not done so. This is an artifact of our methodology and one we hope to address in future work.

4 RESULTS

The quantitative data collected from the experiment consisted of answers to the growth mindset statements, completion times for both control and experimental groups, number of code submission attempts, and the number of attempts on the test case quiz by each student in the experimental group. We first present this quantitative data, followed by insights from the qualitative coding process discussed above.

4.1 Quantitative Results

Table 2 shows the completion rate, mean time and mean number of submissions for the experimental and control groups. "Submissions" refers to a compilation and run against test cases. Participants in the control group completed the programming task at a rate of 52.94% and those that completed it took 23.82 minutes on average with an average of 7.59 code submission attempts. In contrast, participants in the experimental group completed the programming task at a rate of 76.19% and those that completed it took 22.62 minutes on average with an average of 4.48 code submission attempts. From these basic results, it appears that the intervention helped more participants complete the programming task compared to those that did not receive the intervention - the experimental group had a higher completion rate, faster time, and fewer attempts required to complete. However, several of the other data points were inconclusive as we discuss below.

Table 2: Programming task performance of control and experimental groups

	Control	Experimental
Correct completion rate	52.94%	76.19%
Mean time (minutes)	23.82	22.62
Mean code submissions	7.59	4.48

Participants answered the growth mindset questions on a five-point Likert scale and we mapped their answers into a delta change value from before and after the programming task. This revealed a change of +0.308 on average in the control group and +0.000 on average in the experimental group. This seems to indicate that the intervention did not have a positive impact on participants' growth mindset. This could be explained by the fact that the experimental

group displayed a higher level of growth mindset overall from the beginning. Another possible explanation for the increase in growth mindset in the control group (vs none in the experimental), is that perhaps working a problem and completing it in 35 minutes increased participants' confidence in their own abilities. Finally, recent research has begun to question whether the mindset of a student impacts performance, with one recent study (published after we conducted this experiment) concluding that it did not [23].

Finally, only one participant in the experimental group needed to submit the test case quiz more than once. In that one case, the student arrived at the correct answer on the second attempt, but did not complete the programming task. The four other participants in the experimental group that did not complete the programming task all correctly answered the test case quiz on the first attempt. Therefore, this data alone explains very little as there was no correlation between the number of test case attempts and programming task completion.

4.2 Qualitative Results

We tagged data in 1,116 places in the observation notes and post-session interviews, using Loksa et al. [25] and Prather et al. [36] as our theoretical frameworks. As we tagged and coded the data, trends began to naturally emerge from the data, which are explored below.

4.2.1 Experimental Group. Participants in the experimental group who submitted a correct code solution tended to display and verbalize higher metacognitive skills and behaviors. For instance, when asked to describe how he/she normally goes about solving a problem in Athene, participant 27 (P27) described themselves as walking through five of the six stages of problem-solving. Furthermore, all participants in the experimental group, except for one, verbalized a correct understanding of the purpose of the intervention when asked directly about it. P31, who completed the problem in 9 minutes, said that the intervention "help[ed] you understand the problem beforehand, make sure you understand the logic behind it." Most participants verbalized something similar. However, P31 also added that it, "helped me understand that when you dont́ enter anything it would be equal," which is an edge case not explicitly described in the instructions. P43, who completed the problem in 15 minutes, voiced their appreciation for the intervention, saying, "I usually freak out reading the prompt, but doing a test case helped me breathe and know that I know how to do it." P29, who completed the problem in 20 minutes, described how solving a test case immediately after reading the prompt, "made me realize that I didn't read the problem very well because I needed to go back and read it again before I answered the quiz." However, requiring students to solve a test case before coding did not entirely eliminate students from struggling with the *Assumption* metacognitive difficulty (see Table 1). P21, who completed the problem in 30 minutes, correctly solved the test case on their first attempt, and then went on to attempt to solve a similar problem students in CS1 had already seen, "More Even or Odd", where the program must determine if there are more even or odd numbers provided as input. After the eighth submission, P21 re-read the prompt, but continued to attempt to solve the wrong problem, struggling with both the *Location* and *Achievement* metacognitive difficulties. Only after 11 submissions did P21 re-read the prompt a third time and realize their mistake. In the post-session interview, P21 said that the intervention was, "Probably to evaluate the psychology of walking into something and thinking you may not be able to do it... but it helps you know that you can do it", which was not at all the intended purpose.

Participants in the experimental group who did not submit a correct code solution tended toward two behaviors: those that understood the intervention and those that did not. A few participants in this group were able to understand the prompt but were unable to implement a solution, usually becoming stuck on syntax or edge test cases. For instance, P33's code was extremely close to a correct solution, but they did not understand the purpose of the intervention, saying, "it wants to see if I can read a program." Others, however, suffered from various metacognitive difficulties. P39 faced the *Assumption* difficulty, thinking that he/she was solving the problem about even and odds, but discovered this error upon re-reading the prompt after 12 minutes. Other participants suffered from the *Forming* difficulty, such as P42 who struggled to understand the problem, even after correctly solving a test-case, and re-read the instructions five times, each time saying things like, "okay, I'm stupid...that was stupid...dang that changes everything," and, "Oh. Okay. I see now. Well that just makes all that work pointless." P36 also suffered from the *Forming* difficulty, but did not re-read the problem after initially correctly solving a test case. In the post-session interview, he/she said they did not know why they were required to solve a test case before beginning to code. The only participant to incorrectly solve the first test case was P34, who solved it on the second attempt. He/she also struggled with the *Forming* difficulty as he/she repeatedly attempted to count the number of times there were an equal amount of positive or negative numbers before the input had ended. In the post-session interview, he/she said that solving the test case, "tests our problem skills instead of the computer's to see if we can spot the differences first...it made me count how I was formatting the text 'cause my compiler does it for me." It is clear that P34 did not understand the intervention.

4.2.2 Control Group. The control group, who were not asked to solve a test case before they could begin writing code, naturally separated itself during the qualitative tagging and coding process into two distinct groups: those that reconsidered the prompt at least once and those that did not. Ten out of 17 participants did not once re-read the problem prompt, but only one out of that ten, P07, faced any kind of metacognitive difficulty. Even though four of the ten did not arrive at a correct solution, only P07 failed due to an incorrect conceptual model of the problem (a manifestation of the *Forming* difficulty), while the others all struggled with syntax. The other seven participants in the control group who re-read the prompt more than once were often inclined to go back multiple times looking for clues when they became stuck, with one re-reading only once, four re-reading three times, and one re-reading five times. P18, who re-read the problem prompt five times, faced the *Forming* difficulty, and despite so much time spent re-reading the prompt, also faced the *Dislodging* difficulty as re-reading did not appear to change his/her incorrect conceptual model.

However, metacognitive difficulties involving the participants' conceptual models of the problems were not the only ones encountered. For instance, P17, who re-read the prompt three times, submitted an incorrect solution 13 times (none of which included a loop). This was despite the fact that he/she verbalized their confusion as to Athene not recognizing their loop. None of the error messages that he/she encountered in Athene helped him/her to realize that they had failed to correctly adapt and implement a solution, representing a *Location* difficulty. The others in this group all re-read the problem prompt displaying multiple metacognitive behaviors, including forming a correct conceptual model, but faced syntax issues, some of which were overcome.

5 DISCUSSION

The unexpected results from this study lead to some interesting implications and a call for further research. First, those in the experimental group were much more successful at solving the problem than the control group. The qualitative results suggest that, for those in the experimental group, understanding why the intervention took place seems to correlate with success. Most participants verbalized that the intervention helped them think through the problem, including P31 and P29 who said that it helped them understand the problem or address a wrong idea very early in the problem-solving process.

Second, contrary to our expectations, two participants in the experimental group still thought they were solving a different problem ("More Even or Odd", which they remembered from another part of the course) than they actually were. This behavior was also noted by Prather et al. [36] and was the motivating example for designing this particular intervention. We expected this intervention would prevent students from forming the wrong conceptual model because students were explicitly made to choose what the output should be for a given set of input, which included the words "More Positive," "More Negative," or "Equal". Even more perplexing is that there were no negative numbers used in the input of the intervention test case quiz, so even if they thought they were solving the "More Even or Odd" problem, there was no case where answering *Equal* would have made sense if a participant was trying to solve that problem with the input provided. However, we cannot know how many students would have confused the present problem for another, but were prevented from doing so due to the intervention. Based on the overwhelmingly positive qualitative feedback from the experimental group indicating how much the intervention helped them, it is possible that it helped a few escape this issue. One possible explanation for the described phenomenon is that it is an artifact of the intervention using randomly generated test cases and that some sets of input could be interpreted as from the "More Even or Odd" problem. Therefore, one outstanding question is: *How do randomly generated test cases impact studies like this?* Perhaps if students had to solve two or three test cases that were carefully crafted to expose different gaps in their conceptual models, then it might prevent the *Assumption* difficulty, and perhaps help mitigate against the *Forming* difficulty as well. Overall, there was no clear distinction between the groups in terms of qualitative results, with the experimental group showing difficulty with at least four of the five difficulties (*Assumption*, *Location*, *Achievement*, and *Forming*), and the control group demonstrating at least three (*Location*, *Forming*, and *Dislodging*).

Third, we expected that re-reading the problem prompt would be associated with a higher likelihood of solving the problem for those in the control group, but from our results, that is clearly not the case. In fact, it seems like exactly the opposite. One possible explanation for this effect is that the students who already possess some metacognitive abilities only need to read the prompt once before successfully reinterpreting the prompt and creating an accurate conceptual model. This suggests that re-reading the prompt could be an indicator of metacognitive difficulties. This leads us to another outstanding question: *Is there a correlation between number of times a student re-reads a problem prompt and the number and severity of metacognitive difficulties faced?* Understanding this relationship would provide a better look into how and why students lacking metacognitive abilities are struggling to create correct abstractions and conceptual models of the problems they are asked to solve and could provide a fruitful avenue for future automated assessment tool modification.

6 CONCLUSION

In this paper, we presented a think-aloud study of CS1 students in order to determine if specific modifications to an automated assessment tool would improve metacognitive awareness in novice programmers.

Our quantitative data showed promising results, particularly with a higher completion rate, faster completion time, and fewer attempts required to complete for the experimental group. However, data on the pre- and post-test questions, and results on the test case quiz completion were not conclusive. Nonetheless, qualitative data showed that participants in the experimental group who submitted a correct code solution tended to display and verbalize higher metacognitive skills and behaviors, especially regarding correct conceptual models of the problem.

Other indicators showed mixed results, with some participants demonstrating signs of good metacognitive awareness, but others obviously struggling with several of the metacognitive difficulties that novices face when using automated assessment tools identified by [36] (Table 1). There was no clear distinction between the groups, with the experimental group showing difficulty with at least four of the five difficulties (*Assumption*, *Location*, *Achievement*, and *Forming*), and the control group demonstrating at least three (*Location*, *Forming*, and *Dislodging*).

The primary contribution of this work is two-fold. First, our results indicate that this line of research, although not conclusive, warrants future work. Second, we have identified two specific questions for future work: 1. *How do randomly generated test cases impact studies like this?* and 2. *Is there a correlation between the number of times a student re-reads a problem prompt and the number and severity of metacognitive difficulties faced?*

Our future work includes replicating this work at a larger scale, with higher numbers of participants from multiple diverse institutions. We feel this is required as the results from this study are promising but not conclusive. In addition we aim to work towards answering the two questions set out above.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[2] Mousumi Banerjee, Michelle Capozzoli, Laura McSweeney, and Debajyoti Sinha. 1999. Beyond kappa: A review of interrater agreement measures. *Canadian journal of statistics* 27, 1 (1999), 3–23.

[3] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. 2014. Compiler error notifications revisited: an interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 536–539.

[4] Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37, 2 (2005), 103–106.

[5] Brett A Becker. 2016. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 296–301.

[6] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective compiler error message enhancement for novice programming students. *Computer Science Education* 26, 2-3 (2016), 148–175.

[7] Brett A Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 2018 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 640–645.

[8] Brett A Becker and Catherine Mooney. 2016. Categorizing compiler error messages with principal component analysis. In *12th China-Europe Int'l Symposium on Software Engineering Education (CEISEE 2016), Shenyang, China*. 28–29.

[9] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 634–639.

[10] Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the first international workshop on Computing education research*. ACM, 81–86.

[11] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 273–278.

[12] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4.

[13] Karl Anders Ericsson and Herbert Alexander Simon. 1993. *Protocol analysis*. MIT press Cambridge, MA.

[14] Anneli Eteläpelto. 1993. Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research* 37, 3 (1993), 243–254.

[15] Katrina Falkner, Rebecca Vivian, and Nickolas JG Falkner. 2014. Identifying computer science self-regulated learning strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 291–296.

[16] Thomas Flowers, Curtis A Carver, and James Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*. IEEE, T3H–10.

[17] Xun Ge and Susan M Land. 2003. Scaffolding studentsâĂŹ problem-solving processes in an ill-structured task using question prompts and peer interactions. *Educational Technology Research and Development* 51, 1 (2003), 21–38.

[18] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.

[19] Matthias Hauswirth and Andrea Adamoli. 2017. Metacognitive calibration when learning to program. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*. ACM, 50–59.

[20] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.

[21] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 86–93.

[22] Matthew C Jadud. 2006. *An exploration of novice compilation behaviour in BlueJ*. Ph.D. Dissertation. University of Kent.

[23] Antti-Juhani Kaijanaho and Ville Tirronen. 2018. Fixed versus Growth Mindset Does not Seem to Matter Much: A Prospective Observational Study in Two Late Bachelor level Computer Science Courses. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 11–20.

[24] Dastyni Loksa and Andrew J Ko. 2016. The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 83–91.

[25] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1449–1461.

[26] Andrew Luxton-Reilly, Simon, Ibrihim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings of the 2018 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '18)*. ACM, New York, NY, USA. *in press*.

[27] Kathleen M MacQueen, Eleanor McLellan, Kelly Kay, and Bobby Milstein. 1998. Codebook development for team-based qualitative analysis. *CAM Journal* 10, 2 (1998), 31–36.

[28] Murali Mani and Quamrul Mazumder. 2013. Incorporating metacognition into learning. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 53–58.

[29] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 499–504.

[30] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 3–18.

[31] Janet Metcalfe and Arthur P Shimamura. 1994. *Metacognition: Knowing about knowing*. MIT press.

[32] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, Vol. 40. ACM.

[33] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 410–415.

[34] Raymond Pettit and James Prather. 2017. Automated Assessment Tools: Too Many Cooks, Not Enough Collaboration. *J. Comput. Sci. Coll.* 32, 4 (April 2017), 113–121. http://dl.acm.org/citation.cfm?id=3055338.3079060

[35] Raymond S Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 465–470.

[36] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 74–82.

[37] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 74–82.

[38] Ido Roll, Natasha G Holmes, James Day, and Doug Bonn. 2012. Evaluating metacognitive scaffolding in guided invention activities. *Instructional science* 40, 4 (2012), 691–710.

[39] Jeffrey Rubin and Dana Chisnell. 2008. *Handbook of usability testing: how to plan, design and conduct effective tests* (2 ed.). John Wiley & Sons.

[40] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.

[41] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *ACM SIGCSE Bulletin*, Vol. 27. ACM, 168–172.

[42] Michael J Scott and Gheorghita Ghinea. 2014. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education* 57, 3 (2014), 169–174.

[43] Teresa M Shaft. 1995. Helping programmers understand computer programs: the use of metacognition. *ACM SIGMIS Database* 26, 4 (1995), 25–46.

[44] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*. Australian Computer Society, Inc., 87–95.

[45] Dwayne Towell and Brent Reeves. 2009. From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses. (2009).

[46] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).

[47] Christopher Watson and Frederick WB Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 39–44.

[48] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2012. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*. Springer, 228–239.

[49] Jacqueline Whalley and Nadia Kasto. 2014. A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 279–284.