Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools

James Prather
Abilene Christian University
Abilene, TX
jrp09a@acu.edu

Raymond Pettit
University of Virginia
Charlottesville, VA
raymond.pettit@gmail.com

Kayla McMurry, Alani Peters
USAA
San Antonio, TX
kayla.mcmurry,alani.peters@usaa.com

John Homer
Abilene Christian University
Abilene, TX
jdh08a@acu.edu

Maxine Cohen
Nova Southeastern University
Ft. Lauderdale, FL
cohenm@nova.edu

ABSTRACT

Most novice programmers are not explicitly aware of the problem-solving process used to approach programming problems and cannot articulate to an instructor where they are in that process. Many are now arguing that this skill, called metacognitive awareness, is crucial for novice learning. However, novices frequently learn in university CS1 courses that employ automated assessment tools (AATs), which are not typically designed to provide the cognitive scaffolding necessary for novices to develop metacognitive awareness. This paper reports on an experiment designed to understand what difficulties novice programmers currently face when learning to code with an AAT. We describe the experiences of CS1 students who participated in a think-aloud study where they were observed solving a programming problem with an AAT. Our observations show that some students mentally augmented the tool when it did not explicitly support their metacognitive awareness, while others stumbled due to the tool's lack of such support. We use these observations to formulate difficulties faced by novices that lack metacognitive awareness, compare these results to other related studies, and look toward future work in modifying AATs.

CCS CONCEPTS

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

KEYWORDS

Education, CS1, automated assessment tools, HCI, human factors, metacognitive awareness

ACM Reference Format:
James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *ICER '18: 2018 International Computing Education Research Conference, August 13-15, 2018, Espoo, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230977.3230981

ICER'18, August 13–15, 2018, Espoo, Finland

ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3230981

1 INTRODUCTION

Learning how to code involves more than just syntax and data structures; it also requires a mental scaffold around which a learner can correctly place knowledge and begin developing metacognitive awareness [19, 42, 53]. Metacognitive awareness is the ability not only to understand the problem but also to understand where one is in the problem-solving process and to reflect on that state. In his seminal 1945 book, *How To Solve It*, Polya identified four stages that learners move through while solving a math problem, hoping to make learners more explicitly aware of their movement [51]. When Dijkstra attempted to effect this four-stage process in his students, he told them, "Beautiful proofs are not 'found' by trial and error but are the result of a consciously applied design discipline" [13]. In order to be successful in the task of learning programming, novices must adapt these metacognitive strategies [57]. Despite this, most novice programmers lack metacognitive awareness [19], though sometimes the highest-performing novices display some aspects of metacognitive awareness, which could be a clue to their success in such a difficult discipline [8]. Most recently, Loksa et al. [40] investigated metacognitive awareness in novice programmers using a framework similar to Polya's. They identified six specific stages in learning to solve programming problems of which students should be aware: (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.* Loksa et al's approach was to explicitly coach students on these stages and help them identify which stage they were in when they became stuck. In this paper, we investigate the theoretical foundation of Loksa et al.'s proposal in an online learning setting via an automated assessment tool (AAT), looking for ways in which the tool itself could be built to support and help implicitly build a novice's metacognitive awareness. We report on a think-aloud study with CS1 students to observe their interactions with an AAT in order to better understand where AATs presently fail - and where AATs could be augmented - to help novice students build metacognitive awareness in CS1. Therefore, our research question is:

- **RQ:** What difficulties do novices who may lack metacognitive awareness face when using an AAT?

2 RELATED WORK

2.1 Automated Assessment Tools

Because this study's intervention centers on automated assessment tools, it is important to first consider their history and how they are currently designed with regard to supporting metacognitive awareness. In 1960, Hollingsworth created one of the first tools to automatically assess student programming assignments on punch cards [26]. Several hundred automated assessment tools have been created since then with varying levels of adoption [1, 14, 28, 29, 49]. Different tools focus on different aspects of automated assessment: some focus strictly on assessment, others are explicitly built to help novice students, and still others focus on test-driven development [17].

One feature consistently present in AATs that could cause novices to reflect on their location in the problem-solving process, and therefore help to build metacognitive awareness, is compiler error messages (CEMs). CEMs have long been documented as a recurrent source of confusion and frustration to students, and many AATs have been created to address this problem. Traver addresses problems with CEMs, highlighting some of the challenges in improving messages and showing many actual examples of the misleading messages that compilers produce [60]. Murphy et al. were part of a large multi-institution group analyzing debugging strategies of novice programmers [46]. Their observations from class sessions and one-on-one interviews make apparent the frustrations caused by misunderstanding errors in programming code. Finally, Marceau et al. discuss how poor error messages lead to student frustrations, which led Marceau et al. to create their AAT, DrRacket [43]. Furthermore, Marceau observes that some languages used to teach introductory programming, such as Alice [34] and Scratch [41] were created with a goal of protecting students from any possibility of creating syntax errors in their early programs. The rate of error messages has been tied to success through Jadud's EQ measurement [31] and enhanced compiler error messages (ECEMs) have been shown to reduce EQ and similar measures of error rates [6].

Since students so often struggle with understanding CEMs, many creators of automated assessment tools have attempted to enhance the standard CEMs that students receive. One of the earliest examples of an ECEM is seen in CAP, developed by Schorsh in 1995. The intent of CAP was to provide students in an introductory programming course with user-friendly feedback pertaining to syntax, logic, and style errors [55]. In 2012, Watson discussed the tool BlueFix, which applied his principle of adapting the compiler messages to the level of the students [62]. Other examples of enhancing CEMs for novice students include Thetis [22], HiC [24], Expresso [27], Gauntlet [21], a tool by Dy [15], LearnCS! [39], an IDE by Barik [4], and ITS-Debug [11].

A few researchers have reported empirical results on the efficacy of ECEMs. In 2014, Denny et al. reported that there was no statistically significant difference in students' behavior between control and experimental groups [12]. These results seem non-intuitive. In contrast, Becker similarly enhanced CEMs in the automated assessment tool, Decaf, also used for Java programming and found that the enhanced messages actually did change student behavior: after viewing an ECEM, students were less likely to generate the same error in the future [5]. Pettit et al. enhanced CEMs in an automated assessment tool, Athene, used for C++ programming and did not find conclusive results that the ECEMs were more helpful than standard CEMs [50]. Prather et al. took a human-factors approach to redesign the ECEMs in Athene, conducted a mixed-methods experiment, and found that the newly redesigned ECEMs were more helpful than the standard CEMs [52], showing that human-centered design of ECEMs is highly important. Finally, Becker et al. returned to the problem, examined these conflicting results, and found that previous studies were measuring different phenomena. They found a consistent way to explain the seemingly different results on efficacy of ECEMs and reported on another experiment which supported their explanation [7].

Despite the importance of useful feedback messages as cognitive scaffolding to implicitly create metacognitive awareness in novices, and its empirically confirmed helpfulness, most AATs provide only rudimentary feedback for submissions [1, 2]. Reporting on 69 different tools and how they provide feedback, Keuning et al. report that most feedback is on failed test cases, some on failure to compile, and very little about anything else [35]. Most of this feedback is binary in nature (pass/fail). Kyrilov and Noelle report that only providing binary feedback to students leads to lower engagement and higher rates of cheating [37]. It therefore seems that feedback must be enhanced beyond binary pass/fail, but the existing literature has yet to establish a standard design for this feedback.

Hartmann et al. [23] created HelpMeOut, an automated assessment tool that provides students with feedback similar to Denny et al. [12]. HelpMeOut queries a database of similar errors and presents users with examples and how to fix them. This approach contrasts with many others that implement enhanced feedback via expert opinion and not user observation [21, 30]. Furthermore, HelpMeOut's top suggestion is accomplished through crowdsourced voting by students. Hartmann et al. did not attempt to measure whether their AAT helped novice programmers create a better conceptual model of the errors they received or whether it increased learnability for novice programmers. Marceau et al. took a human factors approach to creating DrRacket [44]. They ran mixed-methods experiment and discovered that students were grossly misinterpreting the feedback messages and were confused by DrRacket's highly specialized vocabulary. Marceau et al. postulated that perhaps students do not take the time to read the messages but rather use the presence of CEMs only as an "oracle" that somehow knows how to fix their code; Marceau et al. also suggested that students may read only the code highlights that indicate the necessary change. In following work, they provided a rubric for evaluating the effectiveness of error messages based on student behavior after encountering them [43]. Marceau et al. recommended changes to error messages: simplify vocabulary, be more explicit in pointing to the problem, help students match terms in the error message to parts of their code (e.g. using color coded highlighting), design the programming course with error messages in mind (rather than an afterthought), and teach students how to read and understand error messages during class time.

Several other recent studies utilize aspects of a human factors approach to an automated assessment tool such as the theroy behind error message design [60], design and personification of feedback [38], and eye-tracking to determine if novices read error messages and what they are reading when they do [3].

2.2 Metacognition in Novice Programmers

Introductory courses in programming often focus solely on syntax and data structures, but there is a growing consensus among computer science education researchers that it should also focus on assisting the novice in building a mental scaffold around which they can correctly place knowledge and develop metacognitive awareness [8, 19, 25, 40, 42, 53, 56]. Metacognitive awareness is, simply put, knowing about knowing. Applied to programming, it is not just knowledge of the problem, but knowledge of where one is in the problem-solving process and self-reflection on that state [45].

Incorporating metacognitive awareness into the instruction of novice programmers is rather uncommon, but the subject appears more often in literature regarding intelligent tutoring systems. In 2000, Vizcaíno et al. described the intelligent tutoring system HabiPro [61]. HabiPro included four exercises intended to help students develop good programming habits. The exercises in intelligent tutors can help build mental scaffolding in novices [53], but HabiPro was not designed to build metacognitive awareness. HabiPro is also not an automated assessment tool. A more recent study by Cao et al. reports on Idea Garden, an integrated development environment (IDE) that helps novices by providing mental scaffolding through just-in-time contextual hints [10]. A follow-up study by Jernigan et al. implemented these concepts into a larger prototype and reported that novices in the experimental group required substantially less help than the control group, which did not use the prototype [32]. Finally, Nelson et al. proposed a comprehension-first pedagogy paired with PLTutor, an intelligent tutoring system that aims to help novices better learn meta-programming skills such as code-tracing [47].

Falkner et al. [20] carried out a mixed-methods study that observed self-awareness of learning strategies, such as metacognitive awareness, in novice students in an introductory programming course. Participants engaged in a multi-part reflection process as they worked on programming assessments. Falkner et al. reported that only a few students were able to articulate metacognitive awareness; they therefore recommend instructors engage in targeted explicit cognitive scaffolding to help students develop this skill.

Hauswirth and Adamoli [25] studied CS2 students' metacognitive awareness in two courses by observing what they did in response to explicit coaching on help-seeking and self-assessment behaviors. They found that students engaged in metacognitive activities to varying degrees, but they largely explained this variation as connected to the setup of the CS2 courses. However, they did discover some instances of deeper self-reflection among students in unexpected ways, such as student revision of mastery self-assessments, indicating a student first thought they had mastered a skill and then realized they hadn't. Hauswirth's and Adamoli's study is limited by their data collection method, which was observation of what students did, as opposed to what students thought, even though the study was trying to measure metacognitive awareness. They call for additional work to complement their study by investigating what students are thinking. We attempt to fill this gap through our study's think-aloud protocol.

The most relevant study on promoting metacognitive awareness in novice programmers is by Loksa et al. [40]. As listed above, they identified six distinct problem-solving stages that learners usually progress through sequentially. 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 and required less help. As the literature indicates, then, modified pedagogical approaches and coding 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. Our intention with the present study is to adapt the spirit of these interventions to automated assessment tools that can span this gap.

3 METHODOLOGY

In this paper, we investigate novice programmers' problem-solving abilities by observing them complete a programming assignment using an AAT in order to better understand how an AAT could be built to implicitly increase student metacognitive awareness. In order to understand how AATs might help improve metacognitive awareness in novices, we looked at existing literature for anything that might fit into one of the six stages used by Loksa et al. Only the discussions of feedback in AATs fit that criteria and, when done correctly, could help with stage 5, *implement a solution*. For this study, we chose to use the AAT Athene because of the extensive research already done to enhance its CEMs [48, 50, 52, 59] and its availability for the researchers to further modify. In our previous work [52], we iteratively refined the design of the ECEMs in Athene through two pilot studies and a larger mixed-methods study and reported that the newly-refined ECEMs had a positive impact on student performance. From this work, we are confident that Athene successfully attends to stage 5 by providing helpful and useful feedback to students while they try to write code that can compile. We therefore conducted a think-aloud study [54, 58, 63] to watch novice programmers use Athene and qualitatively analyzed the data in order to understand what difficulties they faced in building metacognitive awareness and how that tool could be further augmented to support users through Loksa et al.'s other five problem-solving stages.

Instead of the regular three hours of classes during week six of the semester in CS1, the primary researcher canceled class and held hour-long one-on-one sessions with each student to provide individualized feedback about their programming process. While meeting for the one-on-one sessions was mandatory, release of information was opt-in, and a different professor than the primary researcher handed out and collected the IRB signature forms while the primary researcher was not in the room. Students were clearly told by the other professor that choosing not to opt-in would not have any effect on their grade and the researchers did not know who had opted in during the one-on-one sessions, as per IRB requirements. All 31 students chose to release their data for this research. Each student met one-on-one with a researcher where the student was observed completing a practical quiz. Students received a programming problem in Athene and had to solve it in a proctored 35-minute time window. Students were asked to verbalize their thoughts while

they solved the problem. In an effort to control for differing development environments and the help students might or might not receive from certain IDEs, students were only allowed to type their code in the default Windows notepad application and could only access compilation and runtime error checking via submission to Athene. After the practical quiz had been successfully completed, or the time had expired, students received detailed feedback on their programming skills and problem-solving process from the researcher.

The general format of the think-aloud study follows the usability testing guidelines found in Rubin and Chisnell [54], 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. Students were then given a very simple task and asked to think aloud so they could get used to verbalizing their thoughts, the observer, and the process, as suggested by [58] and [63]. This simple task was to write a program that would output "Hello, world." This particular task was chosen because it was cognitively easy code to write for any level of student at that point in the semester, so practicing the think-aloud protocol would be manageable.

After completing the warm-up exercise, students were asked to complete the practical quiz 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 was selected because it correlated with course topics at the time and therefore should have been moderately challenging. An additional reason for selecting this problem was that it has been used as an in-class assessment in previous semesters, also with a 35-minute time limit, and a majority of students from those previous semesters completed the problem within the limit. While each student worked to solve the problem, a researcher took extensive notes on what the student did and said. Interactions during the practical quiz between the researcher and student were kept to a minimum per Ericsson and Simon [18].

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 and then moved into ATLAS.ti, a qualitative software analysis package. Tags for the data included Loksa et al.'s six problem-solving stages, users' external reactions to feedback from the AAT, and outward expressions of emotion (verbalizing or demonstrating through body language feeling encouraged, happy, frustrated, or angry). ATLAS.ti determines groundedness as a measure of the relevance of a code within the dataset.

4 RESULTS

In this section, we describe from observation during the think-aloud study how students working in Athene moved through Loksa et al.'s six learning stages. This qualitative data from the think-aloud study will highlight the relevant ways in which AATs, like Athene, fail to help students build metacognitive awareness. We tagged, coded, and grouped the observation notes and interview transcripts into categories by the collaborative agreement of two researchers. This approach allowed larger trends to emerge from the natural groupings of the qualitative data. Specifically, we tagged data in 433 places and from that identified 39 recurring themes as first-order concepts that emerged from our ground-level observations. We then built these 39 themes into five second-order concepts, which are summarized below in the Discussion. To show how we arrived at the second-order concepts, we begin with the raw data of participant experience broken into two categories: students who successfully completed the program within the 35-minute time limit and students who did not. This is followed by a discussion of the data that groups these experiences into broader concepts by using [40] as our theoretical foundation. Student names in this section were changed for the sake of anonymity.

4.1 Students that Completed the Quiz

In the group that completed the quiz, several students finished in under 10 minutes and displayed a similar set of traits. The first trait was a consistent approach to starting the problem. Observation notes report that at the outset, these students, "interpreted the instructions for the problem," and "immediately verbalized a clear conceptual model for the problem." Next, these students followed a similar pattern of thinking through the problem, thinking about how to solve it, choosing a solution (in this case, using a while loop), implementing a solution, and tracing their code with specific test cases in mind. Several students in this group were observed pausing multiple times to think about their chosen solution and their process to solve the problem. Only one student in this group received an ECEM. The student who received this message, Bill, did not read the enhanced portion at first, but read the standard portion, successfully edited his code, and then double-checked his edit by opening the enhanced portion, reading it, and agreeing that his fix was correct. Bill's experience is ideal. Still, by the time these particular students were receiving feedback regarding test cases, they had successfully moved through Loksa et al.'s first five stages of problem-solving and therefore they quickly interpreted any failed test cases and fixed the offending code.

Jane, who took 14 minutes to finish the quiz, first thought through the problem, immediately decided on a solution, and proceeded to create comments about what she planned to do throughout the file and then filled it out with code. This student received one ECEM, did not read the enhanced portion, and immediately successfully edited the code. Her next submission compiled, passed two of the test cases, and failed on the third. She made an edit to her code and resubmitted, receiving the same failed test case message, and then repeated this pattern once more. Finally, after receiving the same test case failure three times, she stopped and carefully walked through her code with specific test cases in mind, found the issue, fixed the code, and finished.

Another student, Patricia, who finished in 18 minutes, read the problem prompt quickly and immediately began attempting to solve it as if it was a different problem students in CS1 had previously encountered that semester ("Even or Odd?": given *n* numbers, compute whether there were more even or odd integer numbers provided as input). She seems to have failed initially to correctly move through problem-solving stage 1, *reinterpret problem prompt*, before moving

on to stage 2, *search for analogous problems*. She correctly chose the "Even or Odd?" problem as analogous, but when moving onto stage 3, *search for solutions*, she chose to use the solution for the "Even or Odd?" problem itself instead of using the problem as the basis from which to form a new solution to a different problem. However, as Patricia began to write her solution, this approach made apparently less and less sense, and she verbalized realizing something was off. She checked the instructions again, but she still didn't seem to understand what was wrong-providing a fascinating case of how forming the wrong conceptual model early on can make it difficult to fundamentally change how one views the programming problem at hand. Finally, after being stuck for a few more minutes, she re-read the instructions a third time and apparently understood. After this, Patricia solved the problem very quickly.

Another interesting group of students who completed the quiz were those who took 30-35 minutes, coming right up against the time limit. Adam, completing the quiz in 30 minutes, read the prompt and immediately verbalized a clear conceptual model of what the problem required and how to solve it. However, Adam ran into extensive issues with syntax and therefore became stuck on stage 5, *implement a solution*. His spoken narration demonstrated that he recognized his deficiency in the particulars of syntax correctness and utilized the enhanced portion of the ECEMs to his advantage, finally solving it on the seventh submission.

Finally, Wayne, who completed the problem in 33 minutes, ran into the same issue as Patricia, confusing the problem for "Even or Odd?" However, Wayne did not recognize his mistake early on. At multiple points in the session, he carefully talked through his algorithm, revealing his incorrect conceptual model. After writing his solution, built for the "Even or Odd?" problem, he encountered one ECEM, fixed the error, and moved on to the final stage, *evaluate implemented solution*, where he failed the first test case. Wayne looked at the expected output compared to the actual output of his program, made an edit, and then passed the next test case. He continued failing test cases, adding to his code to create the right output, and failing the next test case. His code grew longer until he had passed 10 test cases, a process that took just over 30 minutes during which he showed increasing frustration. Finally, at 31 minutes, he re-read the problem prompt and exclaimed, "Oh! Wait! This just hit me that it's doing positive and negative rather than evens and odds. I don't know why that happened," and he very quickly solved the problem. In this case, failing to correctly navigate Loksa et al.'s first few stages of problem-solving apparently led to an incorrect interpretation of the AAT's error messages and an incorrect conception of location in the problem-solving process. By solving compilation problems and working through multiple test cases, Wayne described feeling that he was very close to solving the problem, when he was actually very far away. This story highlights how a lack of metacognitive awareness *almost* kept an otherwise-capable student from succeeding.

4.2 Students that Did Not Complete the Quiz

The 11 students who did not complete the quiz all failed to successfully move through at least one of the problem-solving stages. If the way to a correct solution can be thought of like a path from stage to stage, these students often diverged very early, backtracked frequently, and never returned to the crucial juncture to take the correct path. The most frequent issue these students encountered was a failure to build a correct conceptual model of the problem. Unable or unwilling to spend the time to successfully navigate stage 1, *reinterpret problem prompt*, many of these students searched for analogous problems and solutions to the wrong problem. And, unlike Wayne above, these students never demonstrated recognition that they had the wrong conceptual model. The AAT was not able to alert these students to this failure of metacognitive awareness, allowing them to meander down the wrong path, totally lost until the quiz time had expired.

The most obvious example of a failure to create a correct conceptual model can be seen in the experience of Theo, who spent nearly a third of his quiz time reading and re-reading the quiz prompt. At one point, halfway into the quiz time, the researcher noted that he "just keeps repeating the same phrase from the instructions, 'if the number of positive is greater than the number of negative,' over and over again." Eventually, Theo wrote some code and submitted it, and he received a standard CEM. He spent the rest of his time trying to understand this message. Since the CEM was only responding to a syntax error, if Theo had corrected and submitted his code again, Athene would have begun running his code against the set of test cases. The feedback Theo received was not the feedback that he needed in order to succeed. His time expired while he was re-reading the prompt for the eighth time.

Neil provides a good example of what happens when one fails to navigate each of Loksa et al.'s stages. After skimming the problem prompt, Neil immediately began coding without stopping to think through stage 2, *search for analogous problems*, stage 3, *search for solutions*, or stage 4, *evaluate a potential solution*, jumping right to stage 5, *implement a solution*. This leap was evidenced by his statement after a few minutes, "What I'm wondering is if I need the prompt for input to be in the loop or not," followed quickly by removing that prompt entirely. A minute later, he created two variables and said, "Somehow I'm going to let those represent positive and negative values. I think I'll have to do that in my while loop." At that point in the quiz, his code was structured to accept two integer values and report if they were positive or negative, which is not code appropriate to the correct problem. Minutes later, he said, "I'm going to mentally run through it now," but he did so without any specific test cases. All of these actions show a lack of understanding about what problem he was trying to solve and how to solve the problem he thought it was, as well as an inability to evaluate his own solution. Finally, he submitted his code to Athene and spent the rest of his quiz time working through compiler errors. Slowly working through seven CEMs/ECEMs seems to have provided a false sense of progress to Neil, because his program, even without syntax errors, was very far away from a correct solution.

Thomas successfully navigated stages 1-2, failed to solve stage 3, and was subsequently totally unprepared to move into stages 4-6. Early on, Thomas said things like, "I'm trying to figure out how to . . . that's not going to work," and, "I'm trying to figure out how to make it count the positive ones. I don't know how to . . . that's going to be my issue." He continued tinkering with his code and said, "I just don't know how to see if there's more positive or negative." Thomas' comments reveal that he understood what he needed to do but had great difficulty successfully getting through stage 3, *search*

for solutions. Apparently frustrated and eager for some feedback, Thomas submitted his code, saying, "I guess I'll run it just to see what it will say." His code was syntactically valid and so Athene began running test cases. Once in stage 6, *evaluate implemented solution*, Thomas struggled with the first test case for the remainder of the time, unsure as to how to convert the specified input into the correct output. Near the end of his quiz time, Thomas said, "I feel like I'm close, but I just don't know how to count up positive and negatives. Why is this not working?" The feedback from Athene seems to have given Thomas a false sense of progression through the problem. He described feeling very close, but without finding a solution in stage 3 from which to build his own solution, he was actually quite far from completion. Thomas' experience was almost exactly repeated in the experience of two other students; one, for example, said , "really close to finishing this, I think," when he was quite far away.

Several other observations are worth mentioning as well. A few students said that they usually solve the problem through trial and error. This behavior shows that AATs, such as Athene, allow for submission of code immediately without any assurances that the student understands the problem-they are focused solely on correctness via syntax and test cases. Another issue researchers noticed is that several students seemed to become very frustrated during the quiz, with one student even calling herself and her code "stupid." This suggests that an absence of appropriate feedback from an AAT can contribute to students' feeling lost and frustrated, and potentially forming negative opinions of their work and the discipline. Finally, one positive behavior in this group was displayed by Jenny, who successfully navigated stages 1-3, stalled in stage 4, *evaluate a potential solution*, and finally got out a piece of scratch paper and sketched the flow of the program. This action helped her immensely, and she was able to immediately move on to stage 5, *implement a solution*. Unfortunately, by the time Jenny verbalized the idea to sketch out her solution, her quiz time was nearly over.

5 DISCUSSION

In this section, we will contrast the two groups above in order to answer the RQ: *What difficulties do novices who may lack metacognitive awareness face when using an AAT*? The most glaring inconsistency between those who completed the quiz and those who did not is in the initial formation of a correct conceptual model for the problem, which corresponds to Loksa et al.'s stage 1, *reinterpret problem prompt*. This breakdown shows perhaps the single greatest weakness in modern AATs: the tools merely present the problem and assume that the successful student will eventually conceptualize the problem correctly. Furthermore, there are no measures between viewing the problem and submitting source code to ensure that the student understands what they're being asked to do. As it is, tools like Athene treat every student submission the same, as a solution properly designed and requiring only syntactic corrections for completion. Wayne and Patricia, who both realized their incorrect conceptual model, saw feedback that paralleled the feedback given to multiple students who did not complete the quiz, only these others were not fortunate enough to realize their conceptual errors. It's very possible that these students would have completed the quiz if Athene had somehow prompted them to form the correct conceptual model at the outset. Such a modification would not only benefit the poorer performing students; after all, Wayne completed the quiz, but just barely. It is likely that Wayne would not have taken 33 minutes to solve the problem had he been operating under the correct conceptual model the entire time. Both Patricia and Wayne also illustrate that simply re-reading the problem prompt may not help a student who has formed an incorrect conceptual model, due to the difficulty in dislodging a model once formed.

After forming a correct conceptual model, Jane and several other students who completed the quiz took the time at the outset to build out some scaffolding inside their code by placing comments about how they intended to solve the problem. These students used this technique to navigate stage 2, *search for analogous problems*, by thinking back to similar problems they had encountered, and stage 3, *search for solutions*, by thinking through how they had solved those previous problems, and finally stage 4, *evaluate a potential solution*, by sketching the solution in comments before actually implementing it. This strategy proved to be a helpful way of thinking through an approach before committing to any code. Jenny, who did not complete the quiz, also employed this strategy, but she did so far too late into the quiz time. Meanwhile, many of the students who did not complete the quiz read the prompt (often briefly) and jumped directly to coding, skipping stages 1-4 entirely. This proved disastrous for them as they wandered aimlessly, seeming to hope they would eventually stumble on a solution.

Another important distinction can be drawn in stage 5, *implement a solution*, when considering how some in the incomplete group attempted to work through the received CEMs/ECEMs though they had no idea they had incorrectly navigated all previous stages. Having used Athene for the assigned homework problems in weeks 1-5 of CS1 thus far, these students associated receiving CEMs/ECEMs with being mostly complete, a finding that became obvious during the think-aloud session or in the interviews afterward. This poor sense of location in the problem-solving process ultimately distracted them from the real issue at hand: even if they could get their code to be syntactically correct, their code was not going to solve the problem.

The most-repeated theme in the data also appears in stage 5, *implement a solution*, which was the amount of ECEMs read by students. The participants who did not complete the quiz read nine of the 29 enhanced messages encountered, while the participants that completed the quiz read 23 of the 31 enhanced messages encountered. From the quiz data, reading the enhanced messages seems to correlate with quiz completion among all students that completed the quiz, though it correlates especially strongly for the several students who might not have completed the quiz otherwise. However, we did not control for the so-called "diligent student effect" in this experiment. Students such as Adam, who correctly navigated stages 1-4 but became stuck on stage 5 with syntax errors, heavily relied upon and successfully utilized the enhanced messages to reach a correct solution. Most perplexing is the general behavior of the students who didn't utilize the enhanced messages. One student in particular saw the same ECEM 15 times but never clicked on the enhanced message to expand and read it.

Finally, the experiences of Neil and Thomas can be juxtaposed with the experience of Jane to offer a window into stage 6, *evaluate implemented solution*. When Jane began receiving test case feedback

Table 1: Observed difficulties to metacognitive awareness by novices using AATs

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 leads 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

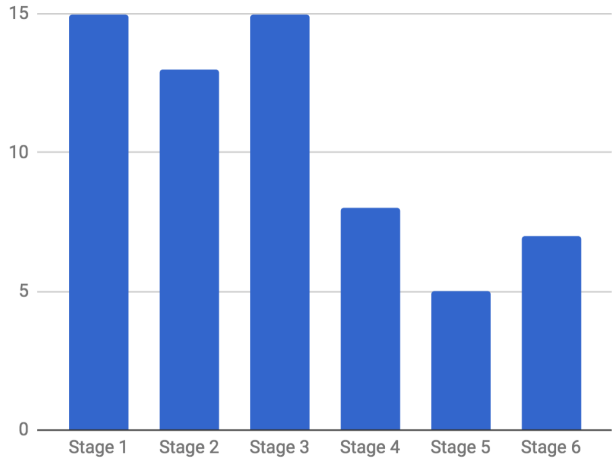

Figure 1: Groundedness of each learning stage within the full dataset.

from Athene, she had already successfully navigated stages 1-5 and was therefore ready to incorporate the feedback accordingly. Because she was solving the right problem, had chosen an approach that could solve the problem, and had correctly implemented the code for her solution, the feedback about failed test cases that she received enabled her to tweak her code and quickly arrive at a correct solution. Both Neil and Thomas, on the other hand, also reached stage 6, but because they had incorrectly navigated stages 1-5, the feedback they received was misleading. Because Athene told them which test cases they had failed, Neil and Thomas expressed that they assumed they should evaluate these feedback messages and that doing so would lead them to a correct solution. Unfortunately, no amount of failed test case feedback would have helped them correct their fundamental misunderstanding of the problem. Reporting that they felt so close to finishing, these two participants apparently never considered that the real issue was their chosen solution.

As noted above, we tagged the data from all 31 student talk-aloud sessions and identified 39 recurring themes as first-order concepts. Six of those themes were Loksa et al.'s six stages, and we used ATLAS.ti to analyze the groundedness of each stage within the full dataset. For reference, the highest level of groundedness of any theme ("helpfulness of ECEMs") was 33, which represents the theme of students reading an ECEM and immediately correcting a compile error. Out of all 39 first-order concepts that showed some level of groundedness, only eight had a higher level of groundedness than the first and third learning stages (which each had groundedness levels of 15); as Figure 1 shows, stage 2 was the next saturated (with a groundedness level of 13). This observation is particularly interesting because "helpfulness of ECEMs" should happen in stage five, implement a solution (groundedness: 5).

In moving from first-order to second-order concepts, we discovered that students often encountered ECEMs when, unbeknownst to them, they were stuck in the first few learning stages by regressing back to an earlier stage from a later stage. Recognizing this pattern helped us to identify the trends in the data leading towards the final two second-order concepts in Table 1, where students encountered ECEMs and mistook it for progress, did not understand feedback that could point them to their incorrect approach to a solution, or were unwilling to backtrack after incorrectly feeling close to a solution.

We went on to group the 39 themes into five second-order concepts that emerged from this research, as summarized in Table 1. The first few difficulties in Table 1 all center around the first three learning stages, which is not surprising given the level of groundedness of the earlier stages compared with the later stages, as shown in Figure 1.

Our results support previous research on metacognitive awareness in novice programmers. Bergin et al. [8] showed that higher-performing students tend to display some metacognitive awareness while lower-performing students display lower or no metacognitive awareness. In our results, students who completed the quiz tended to display some metacognitive awareness, while those who did not complete the quiz tended to lack that awareness. Obviously, determining whether or not a student has some metacognitive awareness depends on whether or not they display it, either verbally or with some behavior. A few students that completed the quiz did so very quickly without saying anything or showing any metacognitive behaviors, such as planning out a solution with comments before coding. However, most students who completed the quiz displayed at least some metacognitive awareness to the researchers. Some students who did not complete the quiz, such as Jenny, displayed some metacognitive awareness but did not do so early enough or consistently enough. Bergin et al. also showed that students with higher intrinsic motivation were more likely to display metacognitive awareness. Our research also confirms this finding. The students who did not complete the quiz showed far less motivation, in part because they seemed to think much less of their programming skills. Several students in the group repeatedly berated themselves out of frustration.

The results of our study should also be brought into conversation with those of Falkner et al. [20]. They found that when teachers included specific exercises in CS1 classes such as design activities, task difficulty assessments, and expectation of iterative coding practices, and when teachers encouraged students to explore alternative designs, students were more likely to develop self-regulated learning strategies such as metacognitive awareness. We found that students who were already aware of their need to practice these skills were more likely to be in the group completing the assessment. Falkner et al. also noted that students in their study frequently skipped design and problem exploration in favor of jumping right into coding. We noted this behavior as well, among both successful and unsuccessful groups, though it was far more frequent among unsuccessful students. The positive scaffolding that Falkner et al. propose and the negative behaviors they observed could be supported and prevented, respectively, if the programming environment that was used implicitly supported metacognitive awareness, which in combination with our research suggests potential direction for modifying AATs to improve student performance.

Finally, Loksa et al. [40] categorize the learning barriers that learners might encounter at a particular stage using the classification system provided by Ko et al. [36]. This classification system describes the different cognitive, environmental, and programming systems barriers that can prevent programming students from finding a solution and solving a given problem. However, only one barrier, *design*, is described as specifically cognitive, while the rest are environmental, programming systems, API interface, or user interface related. The description of the *design* barrier seems to contain several of the metacognitive difficulties we noted above in Table 1, including Forming and Assumption. Ko et al. did not describe the other metacognitive difficulties we found. While the learning barriers presented by Ko et al. can be a useful classification system, it is not particularly equipped to understand the specifically metacognitive difficulties faced by novice programmers. The metacognitive difficulties presented above in Table 1, however, seem to extend the work of Ko et al. by expanding and unpacking their *design* barrier, as well as to offer new insights into the matter.

6 CONCLUSION

In this paper, we have presented a think-aloud study of CS1 students in order to understand the difficulties faced by novices in forming metacognitive awareness when using an AAT. Our analysis shows how successful students mentally augmented Athene while exploring how and why unsuccessful students faltered from Athene's lack of cognitive scaffolding. Unfortunately, this lack of cognitive scaffolding with regard to each of Loksa et al.'s stages is common in most AATs, and so our observations should apply more broadly to a wide range of AATs. The only exception occurs in relation to stage 5, where some tools are already moving to enhance the default CEMs, but even those that have done so are still often lacking in effective human-centered design. The primary contribution of this paper is the identification of metacognitive difficulties that novice programming students often face as shown in Table 1, which can serve to inform future AAT development.

The present study has shown the need for AATs to provide implicit support of metacognitive awareness. Prior research has shown that explicit teaching of metacognitive skills has a positive impact on learning, self-regulation, and growth mindset [40]. In our study, we did not explicitly train students on metacognition because we wanted to determine where AATs presently fail to provide adequate cognitive scaffolding such that a novice can implicitly learn metacognitive programming skills. In this case, we consider metacognitive training implicit because it is built into the tool itself and requires no explicit instruction on learning strategies or stages, and students who use an AAT with built-in metacognitive support would largely be unaware that they were being trained to develop that skill. Our results call for AATs to evolve significantly and move somewhere between their present form and that of intelligent tutoring systems. So what would such an AAT look like?

From the discussion above, it seems that AATs should be modified to provide more comprehensive cognitive scaffolding for novices around which they can appropriately locate their knowledge as they learn. Since a student at a university that uses an AAT will typically complete dozens of problems in a semester, the AAT that is designed to overcome the issues described above will implicitly build metacognitive awareness in novices as they use it over and over again. Such an AAT should help novices work through each of Loksa et al.'s learning stages [40] by confirming that students understand the problem before proceeding, helping them think through previous problems they have encountered, asking them to outline their solution via something like Parsons Problems [33], and requiring students to write their own test cases to submit alongside their code [9, 16], in addition to enhancing the compiler error messages [23, 44, 52]. A more thorough discussion of these suggestions, including empirically testing of each one individually and also testing all of them together, is presently underway and will be reported on in future work.

There are several threats to the validity of this study. First, our observations took place in a laboratory setting and may not reflect actual student behavior and experience. Second, students in the think-aloud study were in a one-on-one setting, were asked to think-aloud, and did not have access to previous code. It is possible that all of these factors increased student cognitive load and, therefore, skewed the results. We attempted to offset the cognitive load concern by adding in the warm-up exercise as suggested by Teague et al. [58]. Third, we acknowledge that coming to firm conclusions in think-aloud studies can be perilous as it can seem like the researchers are narrating the thoughts of participants. However, we only tagged data from students' verbalizations and concrete actions, tying the two together as often as possible, in order to avoid this trap. Finally, the low number of student participants (n=31) is another possible threat to validity. Still, although it is helpful in quantitative studies to increase the number of participants, a higher number would have been prohibitive in conducting an in-depth experiment such as this.

7 ACKNOWLEDGMENTS

Special thanks to Heidi Nobles for her editorial support in preparing this manuscript.

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] José Luis Fernández Alemán. 2011. Automated assessment in a programming tools course. *IEEE Transactions on Education* 54, 4 (2011), 576–581.

[3] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *Proceedings of the International Conference of Software Engineering*. ACM.

[4] 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.

[5] Brett A Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 126–131.

[6] 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.

[7] B. A. Becker, K. Goslin, and G. 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] 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.

[9] Kevin Buffardi and Stephen H Edwards. 2015. Reconsidering Automated Feedback: A Test-Driven Approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 416–420.

[10] Jill Cao, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. 2014. Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers* 27, 6 (2014), 640–660.

[11] Elizabeth Carter. 2015. Its debug: practical results. *Journal of Computing Sciences in Colleges* 30, 3 (2015), 9–15.

[12] 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.

[13] Edsger W. Dijkstra. 1995. Introducing a course on calculi. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1213.html Remarks by Edsger Dijkstra at Department of Computer Sciences, The University of Texas at Austin [Accessed: 2017 08 25].

[14] 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.

[15] Thomas Dy and Ma Mercedes Rodrigo. 2010. A detector for non-literal Java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 118–122.

[16] Stephen H Edwards. 2003. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 148–155.

[17] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 328–328.

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

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

[20] 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.

[21] 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.

[22] Stephen N Freund and Eric S Roberts. 1996. Thetis: an ANSI C programming environment designed for introductory use. In *SIGCSE*, Vol. 96. 300–304.

[23] 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.

[24] Robert W Hasker. 2002. HiC: a C++ compiler for CS1. *Journal of Computing Sciences in Colleges* 18, 1 (2002), 56–64.

[25] 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.

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

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

[28] 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.

[29] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*. ACM, 41–63.

[30] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*. IEEE, T4C–T4C.

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

[32] Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. 2015. A principled evaluation for a principled Idea Garden. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 235–243.

[33] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A mobile learning application for parsons problems with automatic feedback. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. ACM, 11–18.

[34] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1455–1464.

[35] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 41–46.

[36] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.

[37] Angelo Kyrilov and David C Noelle. 2015. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, 122–126.

[38] Michael J Lee and Andrew J Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*. ACM, 109–116.

[39] Derrell Lipman. 2014. LearnCS!: a new, browser-based C programming environment for CS1. *Journal of Computing Sciences in Colleges* 29, 6 (2014), 144–150.

[40] 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.

[41] 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), 16.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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, 163–167.

[47] Greg L Nelson, Benjamin Xie, and Andrew J Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 2–11.

[48] 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.

[49] 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
[50] 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.
[51] George Polya. 1945. *How to solve it: A new aspect of mathematical method* (2014 reprint ed.). Princeton university press.
[52] 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.
[53] 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.
[54] Jeffrey Rubin and Dana Chisnell. 2008. *Handbook of usability testing: how to plan, design and conduct effective tests* (2 ed.). John Wiley & Sons.
[55] 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.
[56] Teresa M Shaft. 1995. Helping programmers understand computer programs: the use of metacognition. *ACM SIGMIS Database* 26, 4 (1995), 25–46.
[57] Judy Sheard, S Simon, Margaret Hamilton, and Jan Lönnberg. 2009. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop*. ACM, 93–104.
[58] 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.
[59] Dwayne Towell and Brent Reeves. 2009. From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses. (2009).
[60] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).
[61] Aurora Vizcaíno, Juan Contreras, Jesús Favela, and Manuel Prieto. 2000. An adaptive, collaborative environment to develop good habits in programming. In *Intelligent Tutoring Systems*. Springer, 262–271.
[62] 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.
[63] 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.