The Role of Self-Regulation in Programming Problem Solving Process and Success

Dastyni Loksa and Amy J. Ko
The Information School • DUB
University of Washington
{dloksa, ajko}@uw.edu

ABSTRACT
While prior work has investigated many aspects of programming problem solving, the role of self-regulation in problem solving success has received little attention. In this paper we contribute a framework for reasoning about self-regulation in programming problem solving. We then use this framework to investigate how 37 novice programmers of varying experience used self-regulation during a sequence of programming problems. We analyzed the extent to which novices engaged in five kinds of self-regulation during their problem solving, how this self-regulation varied between students enrolled in CS1 and CS2, and how self-regulation played a role in structuring problem solving. We then investigated the relationship between self-regulation and programming errors. Our results indicate that while most novices engage in self-regulation to navigate and inform their problem solving efforts, these self-regulation efforts are only effective when accompanied by programming knowledge adequate to succeed at solving a given problem, and only some types of self-regulation appeared related to errors. We discuss the implications of these findings on problem solving pedagogy in computing education.

CCS Concepts
• Social and professional topics~Computer science education • Social and professional topics~CS1

Keywords
Programming, Problem Solving, Self-Regulation, Think-Aloud.

1. INTRODUCTION
Programming problem solving is a complex activity that poses many diverse cognitive demands on learners. As Elliot Soloway argued thirty years ago, expert programmers "have built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them. Students should be taught explicitly about these libraries and strategies for using them." [30]. Students are often left to develop these strategies on their own, and when they fail to do so, they quit [2].

Prior work has investigated a wide range of materials, pedagogies, and techniques for teaching programming problem solving strategies. For example, recent studies have explored worked examples and the effect of sub-goal labels, finding that examples and sub-goal labels can promote greater problem solving success [20,22,23]. Other efforts such as the Idea Garden have investigated strategy hints, giving learners suggestions about how to approach a problem (e.g., divide and conquer), finding that hints can promote independence and self-efficacy [5]. Similarly, Linn & Clancy found that case studies including code and expert explanations can lead to a more integrated understanding of programming process and some gains in problem solving success [18].

While these pedagogies and materials improve learner's *content* knowledge for programming, prior work in the learning sciences literature suggests *process* skills, and in particular *self-regulation*, are equally critical. Self-regulation is the ability to be aware of one's thoughts and actions and evaluate how well they are moving one closer towards a goal [28]. Several studies have investigated self-regulation in learning, finding, for example, that successful learners generate self-explanations of material and use self-explanations to monitor for misconceptions [21]; that self-explanation prompts can improve problem-solving skill and self-efficacy [8]; that high performing CS students use more metacognitive and resource management strategies [3]; and that general metacognitive training can promote improvements in domain-specific skills such as listening and science inquiry [10].

Only a handful of studies have explicitly investigated self-regulation in the context of programming. One of the earliest was conducted by Clements & Gullo, investigating the effect of teaching programming problem solving [7]. They found that teaching programming via Logo, relative to teaching computer use, subjectively promoted greater "reflectivity." Pea and Kurland reviewed this and other work on the effects of learning to code, finding little evidence that learning to code promoted self-regulation or metacognition. However, they they did draw upon learning sciences literature to argue that programming itself *requires* self-regulation for planning programming solutions [24]. This is consistent with more recent work, identifying self-regulated learning strategies [11,12], and showing that programming expertise demands a high degree of self-awareness and self-monitoring [9,17].

Acknowledging that programming requires self-regulation, more recent studies have investigated ways of teaching self-regulation for programming. Bielaczyc et al. investigated the impact of teaching self-explanation, finding that students who received explicit training on self-explanation strategies used these strategies more than those without the training, increasing problem solving success [4]. More recently, Loksa et al. found that combining similar self-regulation instruction with a framework for programming problem solving activities promoted not only greater problem solving success, but also gains in productivity, self-efficacy and growth mindset [19].

While prior work provides compelling evidence that self-regulation is key to successful programming, it leaves several open questions:

ICER '16, September 08-12, 2016, Melbourne, VIC, Australia
© 2016 ACM. ISBN 978-1-4503-4449-4/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2960310.2960334

- To what extent do novices self-regulate when programming?
- To what extent does programming experience in CS1 and CS2 promote self-regulation in programming?
- To what extent is self-regulation related to successful programming problem solving?

In this paper, we investigate these questions, first proposing a theoretical framework for self-regulation in the context of programming. We then present an empirical analysis of novice programmers' self-regulation activities and explore how variation in self-regulation was associated with problem solving success. We end with a discussion our findings on computing education, with several ideas for how to promote self-regulation through teaching.

2. THEORETICAL FRAMEWORK

Lacking an existing theoretical framework of self-regulation in computing, we derive our framework from key self-regulation elements which are common across prior work. Prior work frames self-regulation as the ability to monitor and control one's behaviors, thoughts, and emotions for the demands of the moment, and monitoring progress toward goals [16]. In the context of learning, self-regulation involves *metacognition* (thinking about one's thoughts) [10], *planning* (evaluating progress toward a learning goal), and *motivation* (manipulating one's intrinsic and extrinsic goals to make progress toward learning) [31].

In the context of programming, we propose that self-regulation helps plan and evaluate progress toward writing a program that solves some computational problem. Our hypothesis is that the more a participant self-regulates their programming activities, the more successful they will be at solving the problem.

This hypothesis, however, demands a more granular view of what "progress" means in programming, and how self-regulation is related to progress. Loksa et al. recently defined progress in programming problem solving as six distinct and nominally sequential but iterative activities [19]; we propose these are related to self-regulation as follows:

- *Reinterpreting the problem prompt*. Programming tasks begin with some problem that programmers must interpret and clarify. As with any problem solving, this understanding is a cognitive representation of the problem used to organize one's "continuing work" [13]. The more explicitly one engages in regulating this understanding (e.g., by reflecting on whether their understanding is correct), the more likely they will correct misconceptions of it.
- *Searching for analogous problems*. Programmers draw upon problems they have encountered in the past, either in past programming efforts or even in algorithmic activities from everyday life [14,30]. By reusing knowledge of related problems, programmers can better conceptualize a problem's computational nuances. Learners may self-regulate by being aware of limitations in their knowledge of related problems.
- *Searching for solutions*. With some understanding of a problem, programmers seek solutions that will solve the problem by adapting solutions to related problems or by finding solutions in textbooks, online, or from classmates or teachers [15]. During solution search, learners may monitor the extent to which they have searched and the degree to which the search was satisfactory.
- *Evaluating a potential solution*. With a solution in mind, programmers must evaluate how well it will address the problem. This includes feasibility assessments, mental simulations of algorithm behavior, or other techniques of prototyping before implementation. Self-regulation may help learners to decide whether their evaluation of a potential solution is adequate, or whether they need to more certainty.
- *Implementing a solution.* With a solution in mind, programmers must translate the solution into code using their programming languages and tools. Learners may self-regulate their awareness of working memory limitations and manage prospective memory for future tasks.
- *Evaluating an implemented solution*. After implementing a solution programmers iteratively converge toward correctness, evaluating how well their implementation solves the problem, usually by testing and debugging. Learners may self-regulate their certainty in an implementation's correctness to prevent overconfidence.

Based on these six stages, and prior work on the elements of self-regulation in learning, we identify five types of self-regulation that support programming problem solving:

- *Planning*. Learners should reflect on what their next step in a problem solving process should be (e.g., *did new information reveal a gap in understanding? What tasks remains for an implementation?*) [29]. The more a learner engages in explicit planning, the more successful they should be.
- *Process monitoring*. Programmers who explicitly monitor their progress toward solving a problem are more successful (e.g., *is a sub-goal complete? Is the code sufficiently tested?*) [4,15,17,19]. The more learners monitor when a task is complete, the more successful they should be.
- *Comprehension monitoring*. Learners should monitor their understanding of computational concepts in problems and solutions [19,29] (e.g., *am I confused? Is my understanding of this failure accurate?*). The more aware learners are of their misconceptions, the more successful they should be at correcting them.
- *Reflection on cognition*. Learners should make judgments about the qualities and limitations of their memory and reasoning [31] (e.g., *am I forgetting something? Am I making any assumptions?*). The more aware learners are of their cognitive biases, the more likely they are to correct for them.
- *Self-explanation*. Learners should explain to themselves why they have come to a conclusion or decision, and then use that rationale to monitor their progress [4,21,30] (e.g., *this was the right loop condition because it halts at the end of the list*). The more learners engage in self-explanation, the more they will find flaws in their reasoning.

While all five of these self-regulation activities are likely critical to any kind of problem solving, we suspect that they are particularly useful in programming. This is for two reasons: 1) programming problems are often about abstract computational processes, requiring additional cognitive load to reason about abstract ideas in working memory, and 2) programming languages require precision and completeness, demanding repeated interpretation of the code one has written. Self-regulation may play a key executive control role, facilitating more logical, precise and systematic reasoning about abstract computational ideas, helping a learner to manage their cognition as they navigate a largely invisible solution space.

While self-regulation may be essential some prior work suggesting that teaching it improves problem solving success [4,9,17,19], to what extent do novice programmers engage in self-regulation without explicit instruction? Do these self-regulation skills, however undeveloped, contribute to problem solving success? In the coming sections, we investigate this questions directly.

	CS1 (21 students)	**CS2** (16 students)
Gender	F=11, M=10	F=8, M=8
Age	[18, 18, 27]	[18, 20, 24]
Self-Efficacy	[-3, 2, 8]	[-2, 2, 6]

Table 1. Sample size, gender, age and self-efficacy for each experience group, showing [min, median, max].

3. METHOD

Our target population was students who had enrolled in up to two introductory programming courses (which we will call CS1 and CS2). This ensured exposure to the syntax and semantics of at least one programming language, but minimal experience with problem solving. We recruited participants from lower-division CS and information science classes via email and flyers, ultimately recruiting 37 students. Because CS1 was required for many degrees, but CS2 for only for CS and a few, we viewed these as two separate populations, dividing participants into those who had enrolled in or completed CS1 and those who and enrolled in or completed CS2. Table 1 shows that the groups were balanced across gender, age, and self-efficacy (-8 to 8 on our scale).

When participants arrived, we gathered age, gender, CS course enrollment, and programming self-efficacy, which we adapted from [1] to be language agnostic. Then, to help participants practice thinking-aloud [6], we provided participants with a worked example of a problem consisting of a problem and solution in pseudo-code. We instructed participants to say everything that went through their mind as they read the example and solved problems. If at any point they remained silent for 1-minute they were prompted "remember to think out loud." After participants had time to read and understand the example, they solved a problem that was isomorphic in structure and context. After completing the 1st problem, participants received a 2nd and 3rd problem for practice.

Finally, we gave participants three additional problems to solve one at a time, each without examples. We adapted our problems from prior work that focused on while loop usage [22]. The final three problems consisted of one problem isomorphic to the practice problems, and two context-shifted problems, where the structure of the problem was consistent with prior problems, but the domain was different. Our intent was to provide both familiar and novel problems to investigate variation in self-regulation. Figure 1 shows the final three problems. We instructed participants to write in pseudo-code, focusing on logic over syntax.

ISOMORPHIC PROBLEM

Problem 4: The instructor has now given you a collection of test grades and asked you to calculate the class average for passing grades (those that are 70 or above). Here are all the test grades for the class.

CONTEXT SHIFTED PROBLEMS

Problem 5: Your best friend is a golfer, but is not very good at math. They continue to make errors when adding up scores. You volunteer to write a program that will add up the golf scores and print out the scores for the first nine holes, the second nine holes, and total for the round.

Problem 6: Suppose that a certain group's population grows at a rate of 10% every year. Write a program that will determine how many years it will take for the population to double.

Figure 1. Problem prompts 4-6.

We collected two forms of data. The first was audio recordings of participants' think aloud. We transcribed the recordings and then coded them for 9 types of verbalizations: 4 of the 6 problem solving activities from Section 2 and all 5 of the self-regulation types from Section 2. We based this data collection on best practices of verbal data [6], measures of self-regulation [4,26,28], and theories of problem solving process [19,25]. Table 2 shows our coding scheme for each verbalization type. We merged two problem solving activities—*searching for a solution* and *evaluating a solution*—into *adapting a solution* because they were not observable in our think aloud data and adapting requires finding a solution, and evaluating it. We excluded *implementation* because we did not track participants' editing which are difficult to identify through audio. The 1st author developed the codebook and trained with another researcher, iteratively refining the the coding scheme until reaching consensus. To verify the validity of the scheme each researcher independently coded 10% of the transcripts and then compared each sentence in the transcripts, coming to an 83% agreement across all sentences. The 1st author then completed the remainder of the coding.

The second form of data was the participants' programming solutions. We analyzed solutions for errors by identifying lines of code that would need to be added, removed, or changed in order for the participants' solutions to produce the correct output. We treated lines of code that were unnecessary for a working solution as errors. This analysis was performed only on problems 4-6 because solutions to problems 1-3 were provided in the examples.

Code	**Definition**	**Examples**
Reinterpret problem	Questioning details of the problem prompt or problem requirements.	*"I only need to find the sum?"* *"…it says those that are 70 or above. Does that mean 70% or the number of points?"*
Analogous problem search	Identifying similarities between the current problem and other problems or solutions.	*"It seems like a similar structure of the problem example."* *"So this is like the first question I had."*
Adapt solution	Identifying what needs to change about a prior solution to solve the current problem.	*"This was also like the first one except without the last step of finding the average."* *"This one is a bit different because this time, we have to only calculate the average."*
Evaluate solution	Judging the correctness of code.	*"All right. I'm just going to check the solution."; "So let me just check the for loop."*
Planning	Expressing intent to perform some task, or description of a task participants is doing.	*"I'm going to initialize variables first"* *"I'm just copying the code from the example."*
Process monitoring	Declaring that a programming sub-goal is complete.	*"So that's the end of the for-loop."* *"So I got the first part. Going on to the second. "*
Comprehension monitoring	Reflection about the understanding of code or problem prompts.	*"I don't know, end while means..."* *"And so actually, I don't know how this golf scoring works,"*
Reflection on cognition	Judgments about mental processes, mistakes, assumptions, or biases.	*"I was refreshed from earlier about how to do logical operations within while loop."* *"I read the question wrong."*
Self-explanation	An account of why a decision was correct.	*"So I don't need LCV because, probably because we don't have a list."* *"The average will be zero at first because we didn't add anything."*

Table 2. The 4 problem solving activity codes and 5 self-regulation codes analyzed in participants' think-aloud data, along with definitions and representative quotes from transcripts.

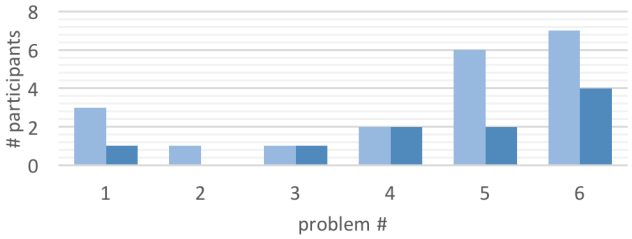

Figure 2. The number of participants who verbalized at least one problem reinterpretation, by problem and experience (CS1→light, CS2→dark).

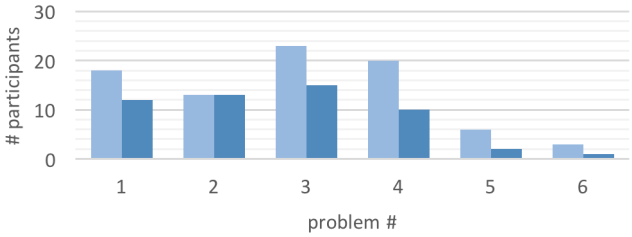

Figure 3. The number of participants who verbalized at least one search for analogous problems, by problem and experience (CS1→light, CS2→dark).

4. RESULTS

In this section, we first discuss the extent to which participants verbalized their problem solving and self-regulation, mirroring the framework in Section 2. Then, we investigate the relationship between self-regulation and errors in participants' solutions. Throughout, we compare the behaviors of participants in the CS1 and CS2 groups. All between group statistical hypothesis tests reported were Kruskal-Wallis tests.

4.1 Problem Solving Process

Here we discuss the four problem solving activities observable in the transcripts, discussing the extent to which participants engaged in them and how they were influenced by self-regulation.

4.1.1 Reinterpreting the problem prompt

Reinterpretation is critical to understanding the nuances and ambiguities in problems [13]. We expected participants to use process and comprehension monitoring to identify knowledge gaps, leading them to reinterpret the problem prompt. Our data showed, however, that very few participants engaged in problem reinterpretation. Of all 37 participants, only 15 verbalized about reinterpreting the prompt. This lack of reinterpretation was consistent across both experience groups: 8 (of 21) CS1 and 7 (of 16) CS2 participants verbalized reinterpreting.

As shown in Figure 2, CS1 participants primarily reinterpreted the context shifted problems, 5 and 6, where they demonstrated difficulty conceptualizing the problem they were attempting to solve. In contrast, the CS2 participants that reinterpreted did so across most of the problems. This suggested a pattern of self-regulation related to experience, but the frequency of reinterpretation verbalizations across all problems was not different between groups (p=0.59, H=2.57).

Participants often began coding without fully understanding the problem, leaving them with knowledge gaps in the problem requirements and causing them to later stop implementation to address the gaps. For example, while implementing a loop for problem 6, P4 (CS1) stopped to question, "*Should I do less than 200? ...doubles? [Should it be] While 100 is less than or equal to 200?*" Only after deciding what logic to use were they able to continue coding. Similarly, P3 (CS1) questioned requirements while coding the output for their solution, realizing that a small detail may invalidate their work: "*do you want me to give you this decimal years, how many years it would take? Because this is a whole different math, I think.*" After resolving this concern, they completed the output and started on the next problem. During this process, comprehension monitoring helped participants identify gaps (e.g., should they use less than, or less than and equal to?) Process monitoring spurred participants into reinterpreting the problem. Stronger self-regulation at the beginning of problem solving may have prevented these disruptive task switches.

4.1.2 Searching for analogous problems

Programmers draw upon knowledge of previously encountered problems to provide insight into new problems [14,30]. We expected that participants would engage in process monitoring and self-explanation to identify when they had found a past problem that might help them build a solution.

Our results showed that participants frequently searched for analogous problems and solutions. Of the 37 participants, 29 verbalized a search for analogous solutions at least once across all problems. Figure 3 shows that this varied by problem and was more prevalent for problems 1-4, where there were prior examples to leverage. Of all search verbalizations, 83% (316 of 380) occurred in this context. Participants may have perceived problems 5 and 6 as entirely new problems, unable to see the deeper structural similarities due to their inexperience.

CS1 participants verbalized searches for analogous problems across many problems, while CS2 participants did not. While the frequency of search verbalizations across all problems was not significantly different (p=0.89, H=0.29), the CS1 participants searched in up to 5 of the 6 problems, relying on prior solutions to solve the problem. In contrast, CS2 participants verbalized searching for only 3 of the 6 problems, with many verbalizing none. This indicates that those with less experience were self-regulating *more*, perhaps due to the problems being more novel to them.

The content of participants' search verbalizations differed by experience. First, CS1 participants tended to explicitly reference examples (e.g. "*...which means I have to combine example one and example two.*" (P26)) while CS2 participants referenced problem details (e.g. "*So it's sort of like the last problem where you need to be keeping track of certain scores.*" (P10)). Another difference was the scope of the analogy identified. CS1 participants often identified similarities about surface features of the solution; for example, P33 identified that their loop should be the same as the one in the example, "*So I think you would just do the same, except you take out everything that's under 70 for this one.*" Similarly, P44 said, "*So the loop termination condition is very similar to the first example.*" CS2 participants indicated the entire solution as being analogous: "*Okay, so this is like the exact same [problem] pretty much with different values.*" (P37). This difference reveals that CS1 participants were self-regulating at a structural granularity, while CS2 self-regulated at a computational level.

4.1.3 Adapting previous solutions

Just as programmers rely on prior knowledge to conceptualize novel problems, they also rely on previous [15]. Self-regulation is integral to this, requiring comprehension monitoring to understand the previous solution and the current problem, while planning the adaptations necessary, all while monitoring their adaptation progress.

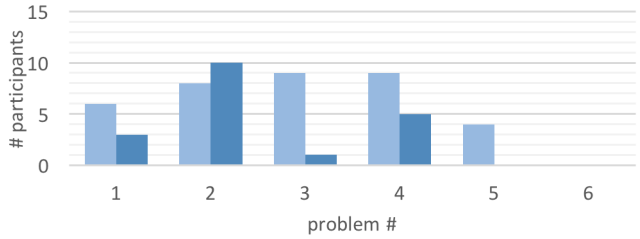

Figure 4. The number of participants who verbalized at least one solution adaption, by problem and experience (CS1→light, CS2→dark).

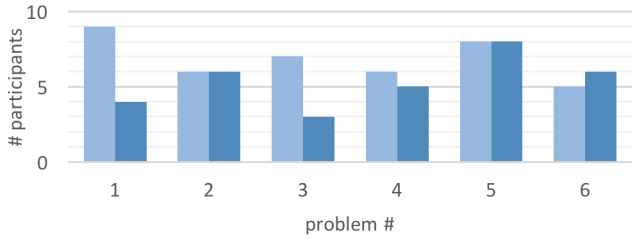

Figure 5. The number of participants who verbalized at least one solution evaluation, by problem and experience (CS1→light, CS2→dark).

Our data shows that although many *searched* for previous problems, only half verbalized *adapting* a previous solution (10 of 21 CS1 and 9 of 16 CS2). However, as Figure 4 shows, frequency varied by experience. CS1 participants tended to verbalize adaptation for more of the six problems, but did not verbalize more frequently (CS1 and CS2 groups had a median of 1 verbalization across all problems, p=0.88, H=0.02).

CS2 participants also appeared to be more confident, suggesting less need for comprehension monitoring. To illustrate, consider P18 (CS1), who said: "*So this one is a bit different because this time, we have to only calculate the average for those students who have passed.*" In contrast, the much shorter, and arguably more confident comment made by P4 (CS2), "*So it's the same problem as example one, it's just the values are different.*"

While the length and tone of the verbalizations for adapting previous solutions varied slightly, the content varied little, with most providing a single high level detail about how the previous solution would need to be changed. Examples include P15 (CS1), who said "*So basically, it's the same thing but now, we're just counting two instead of the sevens*" and P6 (CS2), "*So this time, instead of sevens, we should count the twos.*"

4.1.4 Evaluating solutions

Evaluation of a solution, including analysis and testing, are critical to successfully solving programming problems [17]. We expected participants to engage in comprehension monitoring and process regulation to determine whether to engage in evaluation and determine the quality and level of detail of the evaluation.

Figure 5 shows only about half of participants verbalized evaluation. Overall, 42% (9 of 21) of CS1 participants did compared to 62% (10 of 16) of CS2 participants. However, this difference was not statistically significant (p=0.27, H=1.20).

There were two types of evaluations. Many were short statements that occurred before or just after the mental simulation of code. Those that occurred before announced the *intent* to evaluate. For example, P10 (CS1) completed their solution and said, "*All right. I'm just going to check the solution.*" Similarly, P22 (CS2) finished initializing variables but wanted to verify that they listed the correct values in their array stating, "*Let me to double check*" before reading off each of the values in the problem prompt, verifying they exist in the array. Statements that occurred after evaluation focused on the *result* of evaluation. For example, P10 (CS1) said, "*All right. I am satisfied with this solution*" after tracing their completed solution. P5 (CS2) said, "*I think that's fine*" after briefly looking over their code. These verbalizations likely represented their decision that their evaluation was adequate.

Most evaluations were on entire solutions but some participants evaluated smaller portions of code. For example, P26 (CS1) evaluated the initialization of their grades array: "*I'll just double check to make sure I put them all in correctly*" ensuring that the data was correct. P3 (CS2) verbalized intent to evaluate their loop, "*So let me just check the for loop*", after which they returned to implementing their output. Some participants verbalized their tracing. For instance, P39 (CS1) traced their completed solution while saying, "*Awesome, that should be good. First nine holes zero, second nine holes zero, total score, go through it each time. Print it the first time, print the second time, add them up for a total… Awesome.*" While there were differences between participants, there were no systematic differences between the groups.

Evaluation impacted problem solving by exposing misconceptions and errors and by helping participants gain confidence. For example, P22 (CS2) identified an error: "*Double checking. Yep. Oh, I think we need a print line. Yeah.*" In these cases, participants returned to either reinterpreting the problem to clarify ambiguities, or they returned to code having located a defect. The second outcome was an increase in confidence allowing the participant to continue onto the next sub-problem or problem. For example, after evaluating, P19 said: "*All right. I am satisfied with this solution.*"

4.2 The Role of Self-Regulation

Having discussed the problem solving behaviors that rely upon, and thus indirectly indicate self-regulation, in this section we describe our findings on the role of self-regulation.

4.2.1 Planning

Planning is pervasive throughout programming problem solving, guiding the direction that programmers take and driving the choices of both what to do and when to do it [30]. We expected that few participants would exhibit planning given their inexperience.

In fact, as you can see in Figure 6 the *majority* of participants verbalized planning. Only two did not, both of whom were CS1 participants. For context, one of these participants had slightly fewer errors than the average participant while the other had the 3rd most errors in their solutions among all participants. CS1 participants had a median of 5 planning verbalizations while the CS2 group had a median of 6. However, this was not a significant difference (p=0.17, H=1.88).

When participants verbalized planning, they focused on two topics. First, they spoke about intent to evaluate such as when P10 stated,

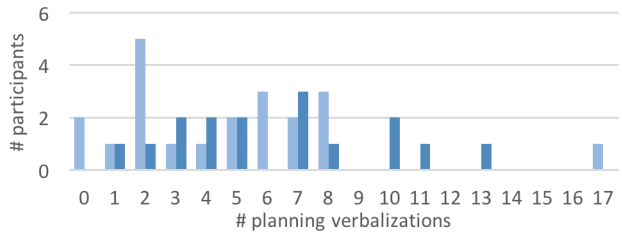

Figure 6. Frequency of participants' planning verbalizations across all problems, by experience (CS1→light, CS2→dark).

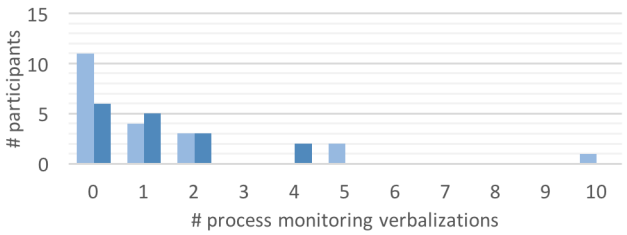

Figure 7. Frequency of participants' process monitoring across all problems, by experience (CS1→light, CS2→dark).

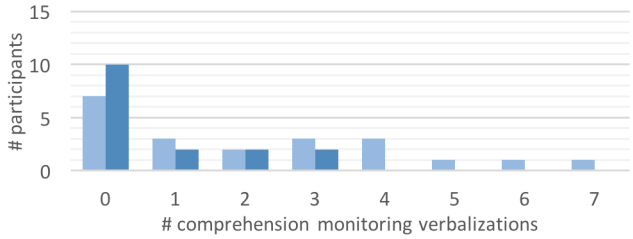

Figure 8. Frequency of participants' comprehension monitoring across all problems, by experience (CS1→light, CS2→dark).

"*I'm just going to check the solutions*" or while already evaluating P16 said, "*Let's go through this one more time.*" Second, they spoke about plans for implementation, primarily for a specific line of code. Examples include P6, who said, "*I'm gonna print out the result*" or P30's realization, "*...and we can do a sum for sum1 right here.*" The more abstract and less granular plans for code included larger sections of code as in P31's comment, "*So after reading this, I think a good first step would be to initialize the variables*" or when P5 decided to write the structure of their if-statement block, "*so first I'm just going to write it.*" There were no discernable differences in the types of planning between the CS1 and CS2 participants.

4.2.2 Process monitoring

Our framework suggests that programmers engage in process monitoring to track their progress through their problem solving process, identifying when goals have been completed, then utilizing planning to identify necessary next steps.

Our data showed that only half of participants verbalized process monitoring and those that did, did it rarely. Figure 7 shows that only 10 of 21 CS1 participants verbalized process monitoring, averaging just 1 verbalization per participant over all six problems. There was one outlier in this group, a 22-year old female, who verbalized about process a total of 10 times across all problems. There were no indications as to why she verbalized process as much as she did and her other self-regulation behaviors were unremarkable, however, she made fewer errors than 63% of participants. CS2 participants had a median of 1 verbalization; not significantly different from CS1 which had a median of 0 (p=0.61, H=0.26).

We observed two types of process monitoring. The most prevalent was a declaration of having completed an implementation sub-goal. This was often verification of completing the initialization of all needed variables, or completion of a loop. For example, P29 said: "*Okay, so I've got the list. I've got the count. I've got LCV. I've got sum.*" or P6's comment about completing the content of a loop: "*So I got the total and the count, so that's the end of the for-loop.*" The second type of process monitoring was when participants declared a solution complete. Examples of this include, "*And printed. I'm on the next task.*" (P1), and "*Yay I'm done (maybe).*" (P22). Both types appeared to help participants segment their process, marking the end of a task and the beginning of planning the next one.

4.2.3 Comprehension monitoring

Our framework suggests that programmers engage in comprehension monitoring to identify knowledge gaps. The more a programmer is aware of their misunderstandings about a problem or a piece of code, or of their own confidence of some given code being correct, the more likely they will make better decisions.

Surprisingly, we found that CS2 participants were much less likely to verbalize comprehension monitoring than CS1 participants. Only 6 of 16 (37%) CS2 participants verbalized comprehension monitoring, compared to the 14 of 21 (66%) CS1 participants. Moreover, Figure 8 shows CS2 participants verbalized significantly less (p=0.03, H=4.70) than CS1 participants with a median of 0 verbalizations per participant to CS1's 2.

There were two types of comprehension monitoring. First, many statements involved participants realizing they did not understand something. For instance, CS1 participant P8 commented, "*I don't know what end while means...*" while reading example pseudo-code, and then proceeded to self-explain, finally coming to an understanding. Similarly, P25 (CS1) acknowledged their confusion after reading an example, "*So I'm a little bit confused.*" Rather than just continuing, their process monitoring facilitated the realization of something that was unclear and they decided to re-reading the example. The other type of comprehension monitoring involved participants absorbing information, often from examples or when attempting to understand a problem. For instance, while reading example code P11 (CS1) said, "*So I think I will say, I 90% understand this method.*" While they acknowledged they did not fully understand the example, they felt their comprehension was sufficient to begin work on a similar programming problem. There was no different in the type of comprehension monitoring made by CS1 and CS2 participants; CS1 participants just verbalized more.

The role of comprehension monitoring was primarily to understand examples or a problem. The majority of verbalizations occurred while reading example solutions, including indicators of understanding (e.g. "*It's very simple and I think people can understand it really well*", P10, CS2) and confusion ("*I'm not sure what the length means?*" (P11, CS1). When participants monitored problem comprehension, they indicated statements of confusion, as in P18 (CS1)'s need for domain knowledge: "*And so actually, I don't know how this golf scoring works. How does the golf scoring works?*" We found no differences in the content of CS1 and CS2 participants' comprehension monitoring.

4.2.4 Reflection on cognition

Our framework proposes that metacognitive reflection helps programmers to be aware of their own thought processes and the limits and biases in their memory and reasoning. Because prior studies characterize metacognition among novices as being rare, we expected few participants to verbalize it during problem solving activities.

Reflection was more common than we expected. Figure 9 shows that CS2 participants tended to reflect (9 of 16, or 56% vs. 9 of 21, or 42% of CS1 participants). CS2 participants had a median of 1 verbalization compared to a median of 0 for the CS1 participants. However, the frequency difference between groups, across all problems, was not significant (p=0.18, H=1.80).

The content of participants' reflections was similar across groups. Some reflections were on process, such as "*I could calculate it by hand, but I don't want to do that*", when P25 (CS1) was contemplating how find the number of 7s rolled on a pair of dice. Another example was P10 (CS2)'s comment, "*I'm thinking about*

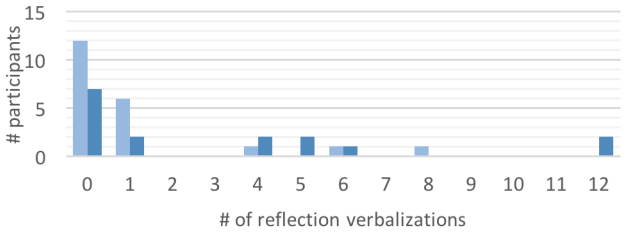

Figure 9. Frequency of participants' metacognitive reflection across all problems, by experience (CS1→light, CS2→dark).

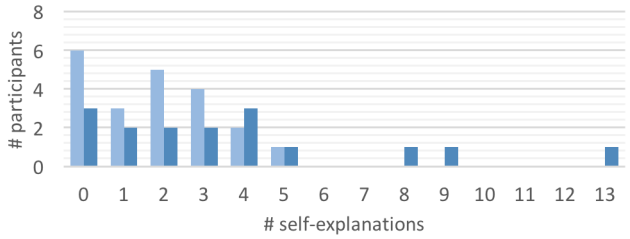

Figure 10. Frequency of participants' self-explanations across all problems, by experience (CS1→light, CS2→dark).

the best way to approach the problem", pausing to consider how to approach solving a problem after reading the prompt. Other reflections concerned confidence. For example, "*And I'm not so sure if this is right*" (P25, CS1), and "*I feel like it's not correct but I'm just going to roll with it*" (P38, CS1). A third type of reflection was when participants identified mistakes, as in "*oh, I forgot to set the rolls.*" (P28, CS2), and "*I feel like I'm wasting mental energy trying to see what scenario is going on when I should be focusing on the essentials*" (P38, CS1). The final type of reflection consisted of reminders, as when P10 (CS2) was trying to establish a process, "*Always need to remember to increment the loop control variable.*"

4.2.5 Self-explanation

In our framework, we suggest that programmers use self-explanation to rationalize decisions they have made and to develop understanding that will influence future decisions. We expected to see participants engage in self-explanation to resolve confusion.

Figure 10 shows that most participants did engage in self-explanation. Overall, 75% (28 of 37) self-explained at some point, with 81% (13 of 16) of CS2 participants self-explaining while only 71% (15 of 21) CS1 participants did. Despite the variation, the frequency of self-explanations across all problems between groups was not significant (p=0.108, H=2.57).

There were three types of self-explanations. Many aimed to increase code comprehension, as in "*Oh wait, no, then it can't be length, because I get to count, all right, count equal zero.*" (P19, CS1), or tracing code for clarification as in, "*And it will not go in two again, which means it will run exactly five times.*" (P28, CS2). Other self-explanations identified participants deciding what code to write. For example, P25 (CS1) was deciding which variables to initialize: "*So I don't need LCV because, probably because we don't have a list.*" P28 (CS2) rationalized about what to write for their loop conditional and said, "*...while loop can use the length, right? Yeah. Because, you have to go through all the items and check it.*"

4.3 Self-Regulation and Errors

In the prior sections we investigated the extent to which participants engaged in self-regulation during problem solving, finding several variations, particularly by experience. In this section, we investigate the extent this variation explained participants' errors.

As we described in Section 2, we measured errors as the smallest number of lines that needed to be added, removed, or changed for a solution to produce correct output. We expected participants with less experience to have more errors in their solutions. Across problems 4-6 (the problems analyzed for errors) CS1 participants' median errors was 6 (with 2 perfect scores on all problems). CS2 participants had a median of 3 errors (with 3 participants receiving perfect scores). For the easier questions, a lack of complexity in the problems, as well as the provided examples, likely contributed to the CS1 participants' ability to craft suitable solutions. On these questions they did not make many more errors than the CS2 group (p=0.34, H=0.88). On the most difficult problem (problem 5 in Table 1), however, the CS1 group made significantly more errors than CS2 (p<.001, H=8.35). This was true despite the problem being only slightly more complex than previous problems.

To investigate the relationship between self-regulation and errors, we built a multiple linear regression model based on several variables. We included gender, programming experience, and self-efficacy, as each tend to effect programming success. We then included frequencies of all five self-regulation types across all problems. This model assumed that verbalizations of each type of self-regulation are indicators of overall self-regulation skill, as opposed to being specific to a problem.

Table 3 shows the resulting model for all participants. We found a significant model ($F(8,28) = 2.66$, p=0.26), with an R^2 of 0.43, with gender a significant factor (p<0.05), with women having more errors in their solutions. Because we found significant disparities in the behavior of participants by experience groups, we also built two separate regression models, one for participants in CS1 (n=21), and one for CS2 (n=16). We included the same factors in these models, excluding programming experience. Table 3 shows the two resulting models for each group. The model was significant for CS1 ($F(7,13)=3.11$, p<0.05), with an R^2 of 0.62, but none of the factors had a individually significant relationship with errors. The CS2 model was *not* significant overall ($F(7,8)=2.232$, p>0.05)—likely due to a small sample size of 16—but there were several large and significant effect sizes in the coefficients that we hypothesized would effect errors (a common rule for judging whether to interpret significant coefficients of a non-significant model). These included a ~3 error *decrease* for each verbalization of comprehension monitoring (p<0.05), a ~1 error *decrease* for each verbalization of planning (p<0.05) and ~1 error *increase* of for each verbalization of self-explanation (p<0.05).

Variable	All participants			CS1 participants			CS2 participants		
	B	*SE B*	*β*	*B*	*SE B*	*β*	*B*	*SE B*	*β*
Gender (M=0, F=1)	3.62	1.58	**0.36***	5.99	2.88	0.52	0.55	1.50	0.09
# CS Courses	-1.49	0.90	-0.30	–	–	–	–	–	–
Self-Efficacy	0.29	0.37	0.12	0.93	0.57	0.31	-0.15	0.29	-0.12
Planning	-0.06	0.23	-0.05	-0.44	0.34	-0.29	-0.79	0.29	**-0.91***
Process Monitoring	-0.64	0.37	-0.26	-0.59	0.45	-0.25	0.26	0.60	0.11
Comp. Monitoring	0.61	0.51	0.23	0.62	0.63	0.23	-2.90	0.99	**-1.09***
Reflection	0.20	0.32	0.13	1.10	0.54	0.41	0.02	0.26	0.03
Self-explanation	0.01	0.42	-0.01	0.11	1.07	0.03	0.98	0.34	**1.19***
R^2		0.43			0.62			0.66	
F		**2.66***			**3.11***			2.23	

**p < 0.05*

Table 3. Three models predicting errors from demographic and self-regulation variables. Unstandardized coefficients (B), standard errors (SE B), and standardized coefficients (*β*).

5. DISCUSSION

Our results show a few trends. First, most self-regulation behaviors were infrequent, inconsistently verbalized, shallow in their application, and often ineffective at reducing programming errors. Among those with more experience, the frequency of planning and comprehension monitoring was related to fewer errors, while more frequent self-explanation predicted more errors. In this section we interpret these results and discuss implications.

The infrequency of self-regulation was quite visible among CS1 participants. They engaged in *searching for analogous problems*, *adapting previous solutions* and *planning*, but showed little depth in reasoning. The infrequency of *reinterpreting the problem prompt* and *evaluation* by CS1 participants was consistent with prior work [27] and their infrequent *comprehension monitoring* and *planning*: they rarely reflected on their understanding of the problem or their code. Table 3 shows the self-regulation they exhibited had little relationship to the errors they made, suggesting that their efforts to self-regulate were simply ineffective.

Similarly, the CS2 group exhibited infrequent and shallow self-regulation. Even the most frequent self-regulation behavior, planning, only occurred about once per problem (Section 4.2.1). CS2 participants also reinterpreted the problem prompt throughout the problem solving process rather than at the beginning, suggesting shallow or absent comprehension monitoring. The lack of process monitoring also plagued CS2 participants. Despite increased experience, they were not very aware of what they are doing or why. That said, the more that CS2 participants engaged in planning and comprehension monitoring, the fewer errors they created (Table 3). This suggests that as learners acquire the necessary knowledge to write programs, self-regulation skills begin to account for differences in success.

It was surprising that self-explanation was not related to success, and in the case of CS2 participants, was associated with *more* errors. This is in direct contrast to prior work, which has shown that self-explanation is a key strategy in problem solving success [4,21,30]. One interpretation is that *verbalizations* of self-explanations are simply more prevalent when participants are struggling: successful participants may have internally self-explained, and done so in a more disciplined manner.

Although we did not study the progression from CS1 to CS2 directly, our results suggest that something is leading to more effective self-regulation. This is in line with Falkner et al.'s work [12] which found that, compared to novices, CS students in their final year of college used more successful self-regulation strategies such as design, testing and problem decomposition. One explanation for this is that CS1 courses are somehow teaching self-regulation and programming problem solving—this is not the case at our institution, but it may arise indirectly through lab sections, TAs, or classroom discussions. Another explanation might be that students who decide to continue to CS2 have independently developed more effective self-regulation strategies.

Interestingly, gender was associated with errors, but only for students CS1 participants. Moreover, this was not explained by differences in self-efficacy. One possibility is that there is some other gender-related factor not in our model; another is that the women who responded to our recruiting were systematically different from the men (we noticed that many women expressed wanting to help future students struggle less than they did).

One implication of these results is on the prior work on self-regulation in computing education. Despite prior studies showing that self-regulation is key in programming expertise [9, 17], and that it can be productivity taught to novices [4,19], our study demonstrates that novices *do* self-regulate, albeit infrequently and poorly. This suggests that efforts to teach self-regulation in CS have a foundation to build upon, but that they may also need to address flaws in students' existing self-regulation behaviors. Our results suggest that these flaws are a lack of *consistent, disciplined* self-regulation during problem solving and few reflections on cognition.

Another implication is that, because self-regulation is only effective with adequate prior knowledge, it may be that the *timing* of teaching self-regulation skills is important. Having disciplined self-regulation skills but lacking adequate programming knowledge may only serve to exhaust and frustrate learners. However, disciplined self-regulation skills may facilitate learners using newly acquired programming knowledge sooner, and more productively.

Our results have several implications for teaching. If self-regulation behaviors are critical to programming success as prior work suggests [9,17], it should be explicitly taught. Prior work has shown that self-explanation [4], and problem solving frameworks [19] can promote success. Our work suggests targeted instruction on specific types of self-regulation—planning and comprehension monitoring—may need further investigation. There are many questions about how these might be taught (e.g., when, how to interleave with syntax and semantics, what pedagogy to use). While only planning and comprehension monitoring had a relationship with errors in our study, other forms of self-regulation might also be taught explicitly. For instance, is instruction on metacognitive reflection [10] beneficial for programmers? How might teaching problem decomposition and reinterpretation effect success? These remain open areas for computing education research.

As with any empirical study, ours had many limitations. First, all studies of this kind could benefit from more data. With a sample size of 37 it was difficult to achieve the power needed to precisely identify effects so many important relationships may have been masked. Also, linear regressions show correlation and not causation; thus, our interpretations may be missing important unseen factors, meaning self-regulation may not cause programmers to craft more successful solutions. Due to typically high variation in programming knowledge, our results were also noisy, further compounding the small sample. While think-aloud protocol is well established for studying self-regulation [6,16], and one of the only mechanisms allowing us to observe cognitive processes, its robustness as a signal varies due to participants' comfort thinking aloud to a stranger. Finally, because participants' knowledge is difficult to measure, our choice of programming problems from prior work led to a slight ceiling effect on errors, hindering our ability to more precisely identify relationships to errors.

Despite these and many other limitations to the validity and generalizability of our results, we view our findings as an important first step in understanding the relationship between self-regulation and programming problem solving. With further research, better instruments, and refined theories, we hope for a future in which teachers not only understand the importance of self-regulation in computing education, but can teach it too.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under grants 1314399, 1240786, 1153625, 1240957, and 1314384. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the views of the NSF.

7. REFERENCES

[1] Askar, P. and Davenport, D. 2009. An investigation of factors related to self-efficacy for Java programming among engineering students. *TOJET: The Turkish Online Journal of Educational Technology* 8.1. http://eric.ed.gov/?id=ED503900.

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

[3] Bergin, S, Reilly, R and Traynor, D. 2005. Examining the Role of Self-Regulated Learning on Introductory Programming Performance. *Proceedings of the First International Workshop on Computing Education Research*, 81–86.

[4] Bielaczyc, K., Pirolli, P. L., and Brown, A. L. 1995. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and instruction*, 13(2), 221-252.

[5] Cao, J., Fleming, S., Burnett, M. M., and Scaffidi, C. 2015. Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers* 27(6), 640-660.

[6] Chi, M. T. 1997. Quantifying qualitative analyses of verbal data: A practical guide. *The journal of the learning sciences*, 6(3), 271-315.

[7] Clements, D. H., and Gullo, D. F. 1984. Effects of computer programming on young children's cognition. *Journal of Educational Psychology*, 76(6), 1051.

[8] Crippen, K. J., and Earl, B.L. 2007 The impact of web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. *Computers & Education* 49, no. 3: 809–21.

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

[10] Fahim, M, and Fakhri Alamdari, E. Maximizing Learners' 2014. Metacognitive awareness in listening through metacognitive instruction: An empirical study. *International Journal of Research Studies in Education* 3, no.3.

[11] Falkner, K, Vivian, R and Falkner, N.J.G. 2014 Identifying Computer Science Self-Regulated Learning Strategies. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 291–96.

[12] Falkner, K, Szabo, C, Vivian, R and Falkner, N.J.G. 2015. Evolution of Software Development Strategies. *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, 243–52.

[13] Greeno, J.G. and Hall, R.P. 1997. Practicing representation. *Phi Delta Kappan* 78(5), 361.

[14] Hoc, J, and Nguyen-Xuan, A. 1990. Language semantics, mental models and analogy. *Psychology of programming*, 10, 139-156.

[15] Ko, A.J., Myers, B. and Aung, H.H. 2004. Six learning barriers in end-user programming systems. *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing,* 199-206.

[16] Koriat, A. 2016. Processes in Self-monitoring and Self-regulation. *The Wiley Blackwell Handbook of Judgment and Decision Making*, 2 Volume Set.

[17] Li, P. L., Ko, A. J., & Zhu, J. 2015. What makes a great software engineer?. *International Conference on Software Engineering-Volume 1*

[18] Linn, M. C., and Clancy, M. J. 1992. Can experts' explanations help students develop program design skills? *International Journal of Man-Machine Studies*, 36(4), 511-551.

[19] Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., and Burnett, M. M. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. ACM Conference on Human Factors in Computing (CHI), to appear.

[20] Margulieux, L.E., Guzdial, M., and Catrambone, R. 2012. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. *ACM International Conference on Computing Education Research*, 71–78.

[21] Michelene T. H. Chi, M. B. 1989. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science* 13(2), 145–82.

[22] Morrison, B. B., Margulieux, L. E., and Guzdial, M. 2015. Subgoals, context, and worked examples in learning computing problem solving. *ACM International Conference on Computing Education Research*, 21-29.

[23] Morrison, B.B., Margulieux, L.E., Ericson, B., Guzdial M. 2016. Subgoals help Students Solve Parsons Problems. *ACM Technical Symposium on Computing Science Education*, 42-47.

[24] Pea, R. D., and Kurland, D. M. 1984. On the cognitive effects of learning computer programming. *New ideas in psychology*, 2(2), 137-168.

[25] Pennington, N. and Grabowski, B., 1990. The tasks of programming. *Psychology of programming*, 307, 45-62.

[26] Pintrich, P. R., Wolters, C. A., and Baxter, G. P. 2000. Assessing metacognition and self-regulated learning. In *G. Schraw, & J. C. Impara (Eds.), Issues in the measurement of metacognition.* NE: University of Nebraska-Lincoln.

[27] Ido, R., Holmes, N. G., Day, J., and Bonn, D. 2012. Evaluating Metacognitive Scaffolding in Guided Invention Activities. *Instructional Science* 40(4). 691–710.

[28] Sawyer, K. R. 2006. Introduction: The new science of learning. In R. K. Sawyer (Ed.) *The Cambridge handbook of learning sciences* (1-18). New York: Cambridge University Press.

[29] Soloway, E., and Spohrer, J. C. 2013. *Studying the novice programmer*. Psychology Press.

[30] Soloway, E. 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.

[31] Winne, P.H. and Perry, N.E. 2000. Measuring self-regulated learning. In P. Pintrich, M. Boekaerts, & M. Seidner (Eds.), *Handbook of self-regulation*. 531-566. Orlando, FL: Academic Press.