©Copyright 2020

Dastyni Loksa

Explicitly Training Metacognition and Self-Regulation for Computer Programming

Dastyni Loksa

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Amy J. Ko, Chair

Katie Davis

Jason C. Yip

Katie Headrick Taylor

Program Authorized to Offer Degree:
Information Science

University of Washington

Abstract

Explicitly Training Metacognition and Self-Regulation for Computer Programming

Dastyni Loksa

Chair of the Supervisory Committee:
Professor Amy J. Ko
The Information School

Programming is one of the most powerful and expressive ways of interacting with computers, but also one of the most challenging to learn. Despite this, people attempting to learn programming often do not receive explicit training or support in developing the mental skills required to succeed. If they are to succeed, learners are often required to independently become self-aware, systematic thinkers while developing and refining strategies to understand and manipulate new abstract concepts in a language they have likely never even seen before.

However, to better support everyone who wants to learn programming, this dissertation presents a *problem solving framework* and a *programming self-regulation framework*. These frameworks give educators and learners the terms and definitions to discuss and reason about the types of behaviors programmers go through to solve programming problems. They also stand as novel domain specific theories of problem solving for researchers to build upon.

To better support learners developing the mental skills necessary for programming, their current skills should be understood. Lacking such an understanding, I conducted two empirical studies investigating the self-regulation of untrained novices, providing a first look into how novices self-regulate while programming, how their self-regulation helps them avoid errors, and where they can use the most support developing critical programming skills.

With an understanding of novices' initial skills, new pedagogical methods should be developed to help learners develop and grow their current skills. To this end, two new pedagogical methods to

support programming problem solving were invented and evaluated. The first of these evaluations demonstrate that not only is explicitly teaching programming problem solving possible, it can help learners become more productive and more independent while boosting their self-efficacy and substantiating their belief that they can become programmers. An evaluation of the second pedagogical method, an online tool delivering a novel form of programming instruction that can support instruction at scale, may help learners achieve more programming success.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . vi

Chapter 1: Introduction . . . 1
- 1.1 The Problem . . . 3
- 1.2 A Solution . . . 3
- 1.3 Approach . . . 4
- 1.4 Definitions . . . 4

Chapter 2: Related Work . . . 6
- 2.1 Understanding Self-Regulation . . . 6
- 2.2 Teaching Self-Regulation . . . 7
- 2.3 Understanding Metacognitive Awareness . . . 7
- 2.4 Modeling Programming Process . . . 8
- 2.5 Existing Problem Solving Frameworks . . . 11

Chapter 3: Theoretical Framework . . . 13
- 3.1 Problem Solving Framework . . . 13
- 3.2 Programming Self-Regulation Framework . . . 16

Chapter 4: Self-Regulation of Novices . . . 18
- 4.1 Introduction . . . 18
- 4.2 Method . . . 19
- 4.3 Results . . . 21
- 4.4 Discussion . . . 35

Chapter 5: Novices' In-Situ Self-Regulation 39
5.1 Introduction 39
5.2 Method 40
5.3 Results 45

Chapter 6: Explicitly Teaching and Scaffolding Metacognition 51
6.1 Introduction 51
6.2 Interventions 52
6.3 Method 53
6.4 Results 61
6.5 Discussion 71

Chapter 7: The Problem Solving Tutor 74
7.1 Problem Solving Tutor 74
7.2 Evaluation 85
7.3 Discussion 93

Chapter 8: Conclusion 96
8.1 Future Work 97

Bibliography 101

LIST OF FIGURES

Figure Number Page

4.1 The number of participants who verbalized at least one problem reinterpretation, by problem and experience (CS1 →light, CS2→dark). 23

4.2 The number of participants who verbalized at least one search for analogous problems, by problem and experience (CS1 →light, CS2→dark 23

4.3 The number of participants who verbalized at least one solution adaptation, by problem and experience (CS1 →light, CS2→dark). 25

4.4 The number of participants who verbalized at least one solution evaluation, by problem and experience (CS1 →light, CS2→dark). 25

4.5 Frequency of participants' planning verbalizations across all problems, by experience (CS1 →light, CS2→dark). 28

4.6 Frequency of participants' process monitoring verbalizations across all problems, by experience (CS1 →light, CS2→dark). 28

4.7 Frequency of participants' comprehension verbalizations across all problems, by experience (CS1 →light, CS2→dark). 30

4.8 Frequency of participants' reflection verbalizations across all problems, by experience (CS1 →light, CS2→dark). 30

4.9 Frequency of participants' self-explanation verbalizations across all problems, by experience (CS1 →light, CS2→dark). 32

6.1 The paper handout and physical token we gave to campers to track their problem solving stage. 55

6.2 Camper E27's final project, showing buttons that link to different interests (left) and content and images (center). Details have been anonymized. 57

6.3 (Main) The Idea Garden panel in the Cloud9 IDE as campers see it when they opened the panel for the first time. (Callout) An example of the Idea Garden decorating the code with an icon. Here, the icon links to the Iteration with For hint. 59

6.4 The total number of strategies mentioned in end-of-day survey responses by campers in each group, sorted by frequency. The experimental group mentioned more strategies than the control group. 63

6.5 The total word count of all end-of-day survey responses by campers in each group, sorted by count. The experimental group wrote more than the control group. . . . 63

6.6 Campers' prescribed task productivity scores by condition, sorted in increasing order. The experimental campers' productivities were typically about equivalent to or higher than the control campers'. 66

6.7 Campers' self-initiated task productivity scores by condition, sorted in increasing order. Experimental campers' productivities were significantly higher than control campers'. Values of zero are not visible. 66

6.8 Cumulative average productivities per project day on both prescribed (light hues) and self-initiated (dark hues) tasks. The experimental group was increasingly more productive than the control group 67

6.9 The campers' median lines of code changes per project day by condition. Experimental campers' amount of code changed was not significantly different from control campers' except on day 8. 67

6.10 Aggregate self-efficacy scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Self-efficacy increased for most campers, but increased significantly more in the experimental group. 69

6.11 Mean self-efficacy in each group for each day of the camp. The experimental campers' self-efficacy increased after the introduction of the intervention (day 2) and ended mildly positive, while the control campers' ended at a neutral level. . . 69

6.12 Aggregate growth mindset scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Growth mindset stayed positive in the experimental group but turned more negative in the control. 70

6.13 Mean growth mindset score across the ten days, by condition. While the control campers' growth mindsets deteriorated sharply when JavaScript was introduced (day 3) and continued to degrade, the experimental campers maintained their existing growth mindset throughout the camp and showed a slight upward trend. 70

7.1 The Problem Solving Tutor interface, showing 1) the programmer, 2) the programming stages, 3) the speech bubble, 4) the code area, 5) the program output area, 6) the navigation controls and current step, and 7) 10 steps from this script's 39 steps. This is script was written for assignment A2a in our evaluation. 75

7.2 A step showing the *excited* emotion. 80

7.3 An assessment step in a PSTutor script, prompting the learner to reflect on what stage the programmer should move to next. 82

7.4 The PSTutor authoring interface, showing the current state of code on the left and the sequence of script steps on the right. 83

LIST OF TABLES

Table Number Page

4.1 Sample size, gender, age and self-efficacy for each experience group, showing [min, median, max] with self-efficacy on a -8 to 8 scale. 19

4.2 Self-regulation study problem prompts 4-6. 20

4.3 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. 22

4.4 Three models predicting errors from demographic and self-regulation variables. Unstandardized coefficients (B), standard errors (SE B), and standardized coefficients (β). 35

5.1 Programming and self-regulation behaviors (adapted from Chapter 3) coded in the journals, with definitions. 41

5.2 The second half of one students' journal for the first homework, showing the codes applied from Table 5.1. This participant was in the *high* coverage cluster. 44

5.3 **Left:** The range of student entries exhibiting each self-regulation or programming behaviors, by assignment, showing higher frequencies of evaluation, planning, process monitoring, and self-explanation than other behaviors and interpret only occurring once per assignment.
Right: The three clusters of behavior (and size of cluster). Each percentage indicates the proportion of students in the cluster who exhibited the behavior at least once in a journal. 46

6.1 The camp schedule, with experimental camp's additions as noted. 55

6.2 Condensed versions of the prescribed tasks given to the campers and the skills that each task required. 58

6.3 Each row defines the barrier and gives an example from a help request, along with the percent of each type of barrier reported by each condition in their help requests. Highlighted cells are the higher of the two proportions. 64

6.4 The barriers from [56] that might be encountered in a particular problem solving behavior. 65

7.1 The metadata required for each step in a PSTutor script. 76
7.2 Steps 13-22 of a script written to support the A4 assignment of the evaluation, showing the speech bubble text. Italicized steps are code edits. 78
7.3 Evidence-based guidelines for authoring PSTutor scripts. 84
7.4 Themes in students' self-reported impact of the PSTutor scripts on their process and self-regulation. 88
7.5 For each of the three assignments, the [minimum, median, maximum] scores out of the maximum possible score, split by students who did and did not use the PSTutor, plus Mann-Whitney U statistics, p-values, and Cliff's delta effect sizes comparing the groups. 91

ACKNOWLEDGMENTS

I would like to thank my advisor, Amy Ko, for her guidance, understanding, and vision. It has been a pleasure working with and learning from her over the past few years. Her insightful advice, both personal and professional, has always been given thoughtfully and as a collaboration seeking to achieve the best outcomes. While I expect that I will always be gleefully basking in the shadow of "the Amy Ko effect" she will always be a role model that I will seek to emulate. I am also deeply indebted to André van der Hoek. Without his efforts to develop an inspiring informatics program at the University of California, Irvine I would not have sought out my undergraduate education. If it were not for him adopting me into his research lab and introducing me to the world of academic research I would not have even considered perusing my Ph.D. nor embarking on an exciting new career path.

I would also like to thank my committee members for their patience, encouragement, criticism, and feedback. Jason Yip always served to provide me a new perspective whether we were discussing learning theories, navigating the east coast weather, or talking about Legos. Despite meeting with Katie Davis only a handful of times during my time working with her, she was always able to provide invaluable insights and provided exactly the resources I needed to move past where I was getting stuck. Unbeknownst to her, her work and valuable resources have also come to me second hand, through her Ph.D. students who I can always count on to have some tidbit of Katie knowledge that turns out to be the breakthrough I needed at the time. Katie Headrick Taylor has also been a strong supporter of my work, eager to hear about new trajectories and lend an encouraging ear. I wish I had taken more time visit the college of education and leverage her knowledge and expertise.

I also offer thanks to all of the lab mates, students, faculty, staff, and friends that have made this work possible including Benjamin Xie, Greg Nelson, Alannah (Val) Oleson, Kyle Thayer, Wanda

Pratt, Caroline (Noth) Pitt, Amanda Swearngin, Amirah Majid, Elizabeth Mills, Rose Paquet, Jake Wobbrock, Michael Lee, Mina Tari, Jin Ha Lee, Calvin (Sega) Apodaca, Will (Rendclaw) Kearns, Marc Schmalz, Jason H. Portenoy, Yim Register, Saba Kawas, Jevin West, Ian King, Abdullah Ali, Matthew Saxton, Matt Davidson, Stefania Druga, Neil Ryan, Andrew Hu, Zak Dehlawi, Roshanak Zilouichian, Harrison Kwik, Maggie (Aithne) Dorr, William Menten-Weil, Dakota Miller, Alexandra Rowell, Meron Solomon, and Polina Charters. Each of these individuals helped me navigate through the tides of the Ph.D. program and inspired my work, or soothed my soul with a heated debate, an interesting conversation, a kind word, helped me grow in one way or another. Thank you all, this would not have been the same without you.

Chapter 1

INTRODUCTION

Programming is one of the most powerful and expressive ways of interacting with computers, but also one of the most challenging to learn [63, 65, 117]. One reason programming is hard to learn is because the list of things people have to learn to program is constantly growing [112]. Learning multiple programming languages [79], programming tools [78], and application programming interfaces [92] are only a few of the countless forms of technical knowledge people need to learn programming.

In addition to technical skills, people need to *believe* they can learn to program, which is challenging in the face of many powerful social forces. Issues such as gender and racial stereotypes about who programmers are [35, 66, 67, 69, 104], teacher bias against students without "the geek gene" [62], and learners' lack of intrinsic interest in computing [35] can severely limit learning outcomes. There are several evidence-based instructional techniques that can substantially improve learning and reduce attrition [90]. However, without careful implementation of these best practices many learners struggle through courses, feeling disoriented, lost, frustrated, and unsupported [53]. Worse yet, recent work has found that introductory CS courses can convince learners that their abilities are fixed and cannot be improved with practice [38, 102], deterring them from further efforts to learn programming.

Another reason programming is hard to learn, and of recent interest in the computer science education research community, is developing *problem solving process knowledge*. Problem solving process knowledge is one's understanding of how to solve programming problems. This includes skills like knowing how to interpret and understand a programming problem [115], envision and adapt algorithms [49], translate those algorithms into a programming language notation [117], verify the implementation actually solves the problem through testing [51], and how to debug a

program when it does not do what was intended [54]. Each of these activities are hard learn on their own. Even more difficult is learning how to decide when to do each of them, and how much of each to do. Perhaps because of these well known difficulties, programming has historically been linked to cognitive ability and problem solving. For instance, there were early claims that the act of programming would **develop** more disciplined thinking processes [26, 80], suggesting that while teaching computers how to think, the programmer also explores their own thinking process [81]. Further work on the topic by Pea and Kurland found little evidence that learning programming promoted disciplined thinking and awareness of process, however, they they did draw upon learning sciences literature to argue that the act of programming does **require** these skills [83].

Once a learner overcomes the difficulties of gaining problem solving process knowledge, they are then confronted with the difficult task of developing the skills to put that knowledge into practice. One general skill set that can help people orchestrate all of these problem solving activities into a successful outcome is *self-regulation* [119]. Self-regulation includes skills like monitoring one's process, reflecting on whether one's process is successful, monitoring one's comprehension of important concepts, and identifying alternative strategies for solving problems. For example, even if someone is still mastering the basics of a programming language, strong self-regulation skills might help them recognize that they do not have a strong understanding of how a *for* loop executes in Python. This might cause them to strengthen their understanding before continuing to write or revise their code. Or, when someone is struggling to diagnose a failure in their program, self-regulation skills might help them recognize that they are floundering and seek expert guidance on how to more productively diagnose the problem. Research consistently shows in both programming [33, 34, 83] and other domains such as math [73] and science [100], that self-regulation skills are strongly associated with problem solving success. Recent work shows, however, that most novices have poor self-regulation skills which are associated with poor problem solving outcomes [44]. While self-regulation is a general skill that applies broadly, it is unclear how easy, and to what extent, successful self-regulation skills in other domains are transferable to programming. What is clear is that without strong self-regulation skills for programming, learners struggle to learn programming [44].

1.1 The Problem

People attempting to learn programming do not often receive explicit training or support for developing the mental skills required for programming. Many opportunities for learning programming focus on the difficult challenge of communicating the vast amount of technical knowledge programming requires, however, this is not the only reason for a lack of training in mental skills. Programming problem solving is a complex activity that poses many diverse cognitive demands on learners. Despite the wealth of research on cognitive activities in other domains such as math and science, it is not yet clear how well those findings apply to the domain of programming. Thus, the cognitive activities for programming are not yet well understood by computer science researchers or educators, making it difficult to successfully develop methods to train learners. Thirty years ago Elliot Soloway argued that 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" [105]. Computer science education has yet to realize this vision. Too often, students are left to develop these strategies on their own, and if they fail to do so, they quit [9].

1.2 A Solution

One solution to support the development of the mental skills necessary for programming is to invent pedagogical methods to explicitly teach learners these mental skills. Thus, my thesis is as follows:

> Novice programmers often lack disciplined programming problem solving, however, explicitly teaching and supporting the development of metacognitive and self-regulation skills can improve learners' problem solving, significantly increase productivity, self-efficacy, and independence, while avoiding growth mindset deterioration.

The goal of this dissertation work is to investigate this claim by exploring learners' existing metacognitive skills as they are applied to programming and using these findings to invent and

evaluate new ways of explicitly teaching and scaffolding metacognitive and self-regulatory skills to support learning programming.

1.3 Approach

This dissertation uses a mixed-methods approach to explore existing mental work and evaluate newly invented frameworks and tools to support the development of mental skills to support learning programming. Each study depicted in this thesis utilizes a different methodology and gathers different types of data to understand the mental work of the participants providing a triangulation of evidence used to evaluate the thesis. All of the studies described herein take a predominately post-positivist theoretical perspective on the work. This is primarily due to a lack of existing knowledge of novices' self-regulation to help interpret results.

1.4 Definitions

There are a number of terms and phrases used in this dissertation that deserve to be clearly defined. This section presents definitions with terms that have no agreed upon definition in literature, or that this dissertation uses in a way that may differ from the expected use.

The first of these terms is *programming*. This dissertation is grounded in the idea that programming is primarily an iterative process of refining mental representations of computational problems and solutions and expressing those representations as code. This definition highlights the cognitive nature of programming and relegates the writing of computer code to an activity that follows from the mental work being conducted by a programmer.

The next two terms that require clarification are *metacognition* and *self-regulation*, which must be addressed in tandem. The literature is unclear about the relationship and boundaries between these two terms. Some models depict self-regulation as a sub component of metacognition while others depict the inverse or suggest they are completely separate cognitive processes. What is agreed upon is that they are both cognitive processes that influence, and are influenced by, many other cognitive processes such as attention and memory. In this dissertation *metacognition* refers to **knowledge** about one's own thinking and knowledge. Metacognition includes *problem*

solving process knowledge such as the mental repository of known problem solving strategies, and identifying which problem solving strategies have been successful or unsuccessful in the past. Metacognition also includes *metacognitive awareness*, which is the **ability** to monitor one's mental state, and the awareness of the mental work currently ongoing.

While metacognition deals with mental knowledge and the ability to monitor mental processes, in this dissertation *self-regulation* refers to the ability to enact **control** over mental processes. Self-regulation includes activities such as evaluating one's understanding of a given problem, the selection of problem solving strategies, stopping to evaluate a strategy in use, and evaluating one's current mental or emotional state.

Chapter 2

RELATED WORK

Research on metacognition and self-regulation are prevalent in various branches of psychology and education, but computer science education is a nascent field and lacks the depth of work on these subjects. This chapter explores five major areas of research: Understanding self-regulation, teaching self-regulation, understanding metacognitive awareness, modeling programming process, and existing problem solving frameworks.

2.1 Understanding Self-Regulation

Computer science researchers are now increasingly conducting research on self-regulation and attempting to understand its role in programming expertise and learning. Most notably, the studies agree that self-regulation is a critical skill for successful programming.

Prior work shows that programming expertise demands a high degree of self-awareness and self-monitoring [31, 59]. In a study of Microsoft software engineers the authors found that experts' systematic and self-reflective methods are a large part of what makes them experts [59]. These types of self-regulation skills manifest as the deliberate systematic practices that expert programmers and teams use to structure their work [85].

It is not just experts who benefit from self-regulation, students also benefit from strong self-regulation skills. Multiple studies have found that the more complex a programming task is, the more that both novice and expert programmers exhibit metacognitive self-regulation behaviors such as explicitly monitoring their progress and reflecting on the effectiveness of their problem solving strategies [31, 56, 93, 102]. In an analysis of teaching and learning programming, Sheard et al. highlighted that self-regulation as a vital set of skills students need to achieve success at programming [103].

Seeking to understand how students navigated learning programming, Falkner et al. investigated the self-regulated learning strategies used by students. Using retrospective survey data, the authors identified examples of successful self-regulated learning strategies used by students. They categorized the strategies as either general strategies which were applied to programming or programming specific strategies. Examples of the strategies they identified include time management, goal setting, and planning. They also found that while explicitly scaffolding strategies helps students develop successful strategies many novices often struggle to engage in the most successful of strategies like problem decomposition and understanding. Additionally, they found that even when they attempt to use these strategies, many do so unsuccessfully. In another study, Liao et al. used interview data to understand areas of student behavior that might help identify characteristics of high- and low-performing students [60]. They found that low performing students were characterized by having used fewer metacognitive or resource management strategies. This finding is consistent with many other studies investigating learning strategy use [10, 86, 88].

2.2 Teaching Self-Regulation

Acknowledging that programming requires self-regulation, Bielaczyc et al. investigated the impact of teaching self-explanation strategies to students. They categorized the self-explanation strategies as "domain-general" strategies which included 1) identifying and elaborating on the relations between the main ideas of a text, 2) determine the form and meaning of code examples, and 3) connect the concepts in the texts to the code examples. They found that students who received explicit training on self-explanation strategies used these strategies more than those without the training, and increased their problem solving success [11].

2.3 Understanding Metacognitive Awareness

Learning to code not only requires effective instruction on syntax, data structures, and abstraction, but also the development of metacognitive awareness [71]. There are many techniques in prior literature for teaching metacognitive skills in other domains. For example, prior studies on problem solving in reading and math taught learners about general limitations and biases in human

learning and memory [29, 100] and provided planning, monitoring, and evaluation checklists [99]. Unfortunately, there is little insight in prior work about how to promote metacognitive awareness in programming. The closest effort is the Idea Garden [19], which helps learners who are stuck by providing deliberately imperfect hints in an IDE and suggests problem-solving strategies (e.g., dividing and conquering, making analogies, and generalizing a solution). There is also some evidence that contextual hints help learners succeed more independently [50] and that scaffolding metacognitive work is beneficial in invention activities [96]. While these are beneficial, they are not designed to promote metacognitive awareness.

2.4 Modeling Programming Process

Programming process is simply the process of solving programming problems. There are several types of media from prior research that have some capacity to model the problem solving process for programming. In this section, I discuss these media types, some of their representative works, and their limitations.

Worked Examples

Worked examples are often used to teach problem solving process in other domains such as math and physics. Subgoal-labeled worked examples from [74, 75] provide a problem, program code that solves the problem, and comments that describe rationale for each part of the solution. These examples can implicitly model the identification of problems, some aspects of planning, and rationale for solutions in comments. However, because they are static code examples they do not explicitly show process. Other than the code comments, usually explaining what a line of code does, these examples hide most of the thinking that occurs during programming relying on learners to make inferences about how a programmer arrived at *that* solution. Programming is highly iterative and by providing only a final solution they hide the authentic iterative process of refining the code into the final solution. This masks all of the revisions that programmers make as well as all of the decisions behind those revisions.

The developers of the ChiQat tutor [2, 40] took an interactive approach to worked examples. The ChiQat tutor provides users with interactive worked examples that allow the learner to step through the implementation of an example often one line of code at a time. These worked examples are not intended to model the problem solving process, but rather to deliver knowledge and rationale about a specific programming concept such as recursion. Despite still not showing any iteration, the ChiQat examples are an improvement over traditional worked examples because the solutions are shown incrementally allowing users to reflect on each line or block of code as it is being written.

Textbooks

Prior research has led to some notable textbooks for explicitly teaching aspects of programming process. For example, *Designing Pascal Solutions: A case study approach* [24] and *Designing Pascal Solutions: Case studies using data structures* [25] were the result of Linn and Clancy's work on case studies [61]. This study showed that students learning with case studies that incorporate expert commentary developed better design skills than students who did not. These texts present the expert commentary and rationale, intermixed with the programming code, in a narrative about how an expert solved a given programming problem. The case studies also use speculative questions, prompting users to reflect on choices and predict consequences and further supporting reflection and metacognitive awareness. While these books do a great job of modeling process, it may still be difficult for learners to emulate the programming process depicted for a few reasons. For instance, each case study presented in the texts are a full chapter with a median of 44 pages. Despite the lengthy page count, these case studies are still very dense and require learners to be both highly motivated and focused. The information presented in each case study is also highly specific to the problem presented which can make it difficult to transfer rational from the case studies to novel problems. These case studies far surpass any programming worked examples in both content and quality, but they are not without their drawbacks.

Another notable and widely used book is *How to Design Programs* (HtDP), which also aims to teach process. Recent work has used HtDP as a basis for studying how students use prior knowledge to create programming plans [36, 37], demonstrating its potential to teach process. These books

model planning and evaluating solutions as well as iteration at an architectural level, but not an algorithmic level. Of course, because textbooks are static, they cannot easily show a program being edited at a low level of granularity or easily illustrate the effect of edits on program output. The process depicted also prescribes a single approach to writing programs and lacks support for students who might make missteps along the way.

Programming Demonstrations

There are many contexts in which instructors provide demonstrations of programming process, or professional software developers livestream themselves programming to help others learn [118]. Unlike worked examples and textbooks, demonstrations have a strong potential to show process, because they are inherently temporal and focus learners' attention on a programmer, their decisions, and their actions. However, to scale them, they must be recorded, which imposes challenges for modeling self-regulation: the programmer has to perform a prescribed demonstration well, and comprehensively think aloud while programming. If they make a mistake, recordings are hard to edit, which may be important as instructors learn how to improve their explanations. Prior work suggests that learners also have challenges when browsing, skimming, and navigating instructional videos [82], while other studies have identified difficulties in sustaining engagement in videos [42].

Programming Tutors and Tutorials

Prior work has also explored intelligent tutoring systems that teach various aspects of programming. For instance, the original LISP tutor focused on basic functions in LISP and how to compose them into basic algorithms [3]. Similar tutors teach specific knowledge, such as how specific programming languages execute programs [79] and how to trace programs [46, 116]. There are hundreds of tutorials online that present content for learning and practicing basic programming language constructs and algorithms [52]. While all of these have the capacity to model problem solving process, they generally do not. Instead they provide static code examples, formative assessment of specific programming language knowledge, and in the most advanced cases targeted

feedback about misconceptions about this knowledge.

Some intelligent tutors in other domains such as math have embedded self-regulation prompts into problem solving. For example, Long et al. embedded self-assessments into a math tutor, which helped promote better problem solving [64]. Other tutors have provided feedback on self-regulated learning strategies in the context of math problem solving [1, 68]. While these approaches have been shown to effectively scaffold self-regulation skills during problem solving, they have not been explored in open-ended programming contexts and there have been few efforts to assess if self-regulation skills transfer to future contexts.

2.5 Existing Problem Solving Frameworks

There are multiple general problem solving frameworks or problem solving frameworks from other domains, however, none of them are specific to programming. One example of a general problem solving framework is presented in the book, *The IDEAL Problem Solver*. The IDEAL framework identifies five steps of problem solving. The first is to *identify* problems and opportunities so you understand what the problem is and frame it as an opportunity to be creative. The second step is to *define* goals so you understand what the wanted outcome is. The third step is *explore* possible strategies to solving the problem which may require reanalysis of your goals so you know what your options are. The fourth step is to *anticipate* outcomes and act, stating that you must consider the ramifications of the selected strategies and outcomes and actively test them through prototypes or simulations to understand what those ramifications might be. The final step is *look and learn* suggesting that we should reflect on our strategies and their outcomes and learn from the process of solving the problem. These steps certainly provide some guidance about how to approach problems, but they are presented very abstractly so that they can apply to a wide range of problems. The steps also require many concrete examples to be able to understand and employ them successfully. It can be difficult to see how these steps directly apply to programming. For instance, because programming is very abstract, when exploring possible strategies to solve a problem a programmer may need to write code to understand if these possible strategies might work. If they do work then the problem may be solved on step three. If the strategy doesn't work, is it unclear if that is

because because the code isn't structured correctly or if the strategy itself is not a good match for the problem. Because of these, and other, confounds a very general problem solving framework is ill suited for the abstract and iterative nature of programming.

An example of a domain specific framework is Polya's mathematical problem solving framework from his book *How to solve it* [89]. In this book Polya provides a framework of four steps to solving problems: Understand the problem, make a plan, carry out the plan, and look back on your work. Accompanied with each of these steps are rules of thumb on how to make progress on the steps. These included strategies such as focusing on the unknown, solving similar problems first, and working backwards. This framework and accompanying strategies greatly advanced problem solving in mathematics education and problem solving in general. However, little is known about how to apply this framework to programming or how the structure of a programming problem might differ from that of a mathematical problem. Additionally, programming strategies differ from mathematical strategies. For instance, working backwards in math might mean working from a correct answer and deriving the steps necessary to come to that solution. In programming, a correct answer is the working code, thus, the problem is already solved. To achieve benefits similar to those that Polya's framework had on mathematical problem solving, computer science education requires a problem solving framework specifically tailored to the domain of programming.

Chapter 3

THEORETICAL FRAMEWORK

Lacking existing domain specific theoretical frameworks for conceptualizing how one solves programming problems, this dissertation contributes two complementary frameworks: The *problem solving framework* and the *programming self-regulation framework*.

3.1 Problem Solving Framework

The problem solving framework describes a set of six nominally sequential behaviors that literature on the psychology of programming suggest are essential to successful programming. Programmers frequently engage in and revisit these behaviors as they iteratively implement a solution and discover knowledge about the problem and solution that was not initially apparent. These six behaviors are:

- *Reinterpret problem prompt*. Programming tasks typically begin with some description of a problem which programmers must understand, interpret, and clarify. As with other forms of problem solving, this understanding is a cognitive representation of the problem used to organize one's "continuing work" [41]. The more explicit this interpretation process, the more likely a programmer will overcome ambiguities in the problem [93]. The less explicit this process the more likely a programmer will misinterpret the problem and be working to solve something that was never intended.

- *Search for analogous problems*. Programmers draw upon problems they have encountered in the past, either in past programming efforts or perhaps in algorithmic activities they have encountered in life (e.g., sorting a stack of books or searching for one's name in a list) [45]. By reusing knowledge of related problems, programmers can better conceptualize a problem's computational nuances. Without explicitly drawing on this knowledge, programmers may

spend a lot of time and effort seeking a novel solution to what may be very familiar problems.

- *Search for and adapt solutions*. With some understanding of a problem, programmers seek solutions that will satisfactorily solve the problem by adapting solutions they have used in the past or by finding solutions in textbooks, online, or from classmates or teachers [14, 56].

- *Evaluate a potential solution*. With a solution in mind, programmers must evaluate how well this solution will address the problem. This includes actions like feasibility assessments, mental algorithm simulations, or other techniques of sketching or prototyping a solution before implementing it [55]. Without evaluating potential solutions programmers may waste time writing code and integrating it into their program only to find that it does not actually solve their problem.

- *Implement a solution*. With an acceptable solution in mind, programmers must translate the solution into source code using their knowledge of languages and tools.

- *Evaluate implemented solution*. After implementing a solution, programmers iteratively converge toward a solution by evaluating how well their current implementation solves the problem. This typically involves software testing and debugging [56, 93].

While this framework holds many similarities to numerous frameworks for problem solving in specific domains such as Polya's mathematical problem solving framework [89], and general problem solving techniques like Bransford and Stein's IDEAL problem solving method [16], the problem solving framework has been refined with language and definitions that are particularly useful for those learning programming. It was designed to provide learners with the *problem solving process knowledge* to support building robust *metacognitive awareness* and the language to discuss and reflect on their process. The design follows recommendations that effective metacognition instruction should 1) provide an abstract understanding of a domain's problem solving knowledge, 2) teach a domain's goal structure, and 3) provide incentives to learn from and avoid common metacognitive errors in the domain [95]. This framework meets the first recommendation by

providing knowledge of the range of activities that programmers engage in to solve problems. The second recommendation is met by imparting ways that programmers converge toward a solution. Presenting the behaviors in a nominal order provides novices with a goal for the next behavior they might engage in based on their current problem solving state. The framework meets the third recommendation by supporting the reflection on, and regulation of, strategies. Having an awareness of the framework, novices can use it to reflect on their current problem solving state, evaluating if they should engage in any of the previous behaviors or if they are ready to progress to the next behavior.

The problem solving framework is applicable to any granularization of a problem and depicts only high-level behaviors that apply to all types of problems. It can be used to situate a programmer in beginning work on a high-level project concept as well as help guide solving a problem that comprises a single line of code. Because the framework is focused on its applicability to all problem types the list of programming behaviors is not exhaustive. Future work should identify and situate behaviors that occur in specific cases such as defect location during debugging or program comprehension behaviors during program analysis.

It is important to note that while the problem solving framework describes the behaviors of programmers, the presentation of the framework should be adapted for its intended use. Because the framework describes mental behaviors, in practice it is not always clear where one behavior ends and another begins. Much of this work, even in novice programmers, is often proceduralized and not consciously engaged in. For instance, during a *search for analogous problems* one might also be *searching for and adapting solutions*.

As an example, searching for and adapting solutions may only *not* be coupled with the search for analogous problems when the search results in a similar problem but no solution. For instance, consider searching for the answer to a problem in online forums and finding the same question you have being asked, but that question having no responses. This would then force one to actively engage in seeking a solution or a continued search for an analogous problem that is presented with a solution. The distinction between *searching for and adapting a solution* and *evaluating a potential solution* is similar. Often, when adapting a solution one may automate the evaluation of the solution

to validate that the adaptation was successful. Because of these vague boundaries, it may be more suitable to present learners with a list of behaviors that combines *search for analogous problems* and *searching for solutions* or combines *adapting solutions* with *evaluating a potential solution*. Alternatively, using the the problem solving framework as presented may allow researchers a more detailed analysis of the behaviors programmers may be engaging in.

3.2 Programming Self-Regulation Framework

This dissertation posits that, in the context of programming, self-regulation helps plan and evaluate progress toward writing a program that solves some computational problem. Unfortunately, while there are many self-regulation theories and frameworks (e.g. [94, 107, 111]) none are specific to computing and it is unclear how they might be applicable to computing. There is, however, prior work on specific aspects of self-regulation in the context of programming from which a framework can be built. Thus, the *Programming Self-regulation Framework* is derived from key self-regulation elements which are common across prior work in computing. This framework identifies 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 remain for an implementation?) [106]. 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?) [11, 56, 59]. 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 [106] (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 [114] (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 be able to explain to themselves why they have come to a conclusion or decision, and then use that rationale to monitor their progress [11, 22, 105] (e.g., this works because it goes through each of the values and halts at the end of the list). The more learners engage in self-explanation, the more they may 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.

Chapter 4
SELF-REGULATION OF NOVICES

In this chapter I describe the first of two studies investigating the self-regulation of novice programmers. Lacking an existing understanding of novice programmers' self-regulation skills, this study[1], and the study described in Chapter 5, will establish an initial baseline expectation of novice programmers' self-regulation skills.

4.1 Introduction

Prior work offers many insights into the effects of self-regulation on novices' learning, but they do not provide the full picture. Studies have investigated self-regulation in learning, finding that successful learners generate self-explanations of material and use self-explanations to monitor for misconceptions [72], that self-explanation prompts can improve problem-solving skill and self-efficacy [28], and that successful CS students use more metacognitive and resource management strategies [10]. While self-regulation may be essential for learning, prior work also suggests that teaching it can improve problem solving success [11, 31, 59]. However, none of these works seek to understand novices' self-regulation before attempting to influence it.

It is important to understand learners' current skills and knowledge when attempting to teach them [15]. Thus, when attempting to teach self-regulation skills it is important to consider the extent to which novice programmers engage in self-regulation without explicit instruction and if their self-regulation skills, however undeveloped, contribute to problem solving success. Prior work provides compelling evidence that self-regulation is key to successful programming, but it leaves several open questions:

---

[1] First published: Dastyni Loksa and Amy J Ko. The role of self-regulation in programming problem solving process and success. In Proceedings of the 2016 ACM Conference on International Computing Education Research, pages 83-91. ACM, 2016.

- 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 novices' self-regulation related to successful programming problem solving?

In this chapter I present a study investigating these research questions. I describe a laboratory study of novice programmers' self-regulation. I then follow with an empirical analysis of novice programmers' self-regulation activities while programming and explore how variation in self-regulation was associated with problem solving success. I end the chapter with a discussion of the findings and several ideas for how to promote self-regulation through teaching.

4.2 Method

The target population for this study was students who had enrolled in two sequenced 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 in the context of programming. 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 only for computer science, we viewed these as two separate populations, dividing participants into those who had enrolled in or completed CS1 and those who had enrolled in or completed CS2. Table 4.1 shows that the groups were balanced across gender, age, and self-efficacy. We gathered programming self-efficacy using a survey adapted from [4] to be programming language agnostic. To

	CS1 (N=21)	CS2 (N=16)
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 4.1: Sample size, gender, age and self-efficacy for each experience group, showing [min, median, max] with self-efficacy on a -8 to 8 scale.

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.

Table 4.2: Self-regulation study problem prompts 4-6.

help participants practice thinking-aloud [21], we provided participants with a worked example of a 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 we prompted them to "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 [76]. The final three problems were one isomorphic problem, 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. Table 4.2 shows the final three problems. We instructed participants to write in pseudo-code and to focus on logic over syntax.

We collected two forms of data for this study. The first was audio recordings of participants' think aloud. We transcribed the recordings and then coded them for 9 types of verbalizations. We coded for the *Reinterpret problem prompt*, *Search for analogous problems*, *Adapt solutions*, and

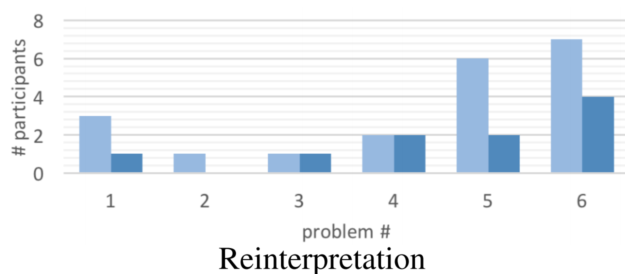
Evaluate implemented solution problem solving behaviors from the problem solving framework detailed in Section 3.1 and all 5 of the self-regulation types from the programming self-regulation framework detailed in Section 3.2. We based this data collection on best practices of verbal data [21], measures of self-regulation [11, 87, 98], and theories of problem solving process [84]. Table 4.3 shows our coding scheme for each verbalization type. We merged two problem solving behaviors — 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. I developed the codebook and trained with another researcher, iteratively refining the the coding scheme until reaching consensus. To verify the reliability 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. I 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.

4.3 Results

In this section, I discuss the extent to which participants verbalized their problem solving and self-regulation, mirroring the frameworks in Chapter 3. 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.

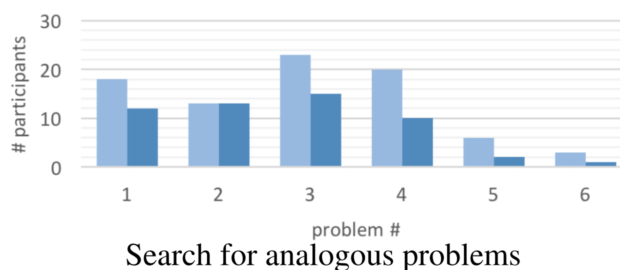
Code	Definition	Examples
Reinterpret problem	Questioning details of the problem prompt or problem requirements.	"I only need to find the sum?"; "...it says those that are 70 or above. Does that mean 70% or the number of points?"
Analogous problem search	Identifying similarities between the current problem 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 4.3: 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.



Reinterpretation

Figure 4.1: The number of participants who verbalized at least one problem reinterpretation, by problem and experience (CS1 →light, CS2→dark).



Search for analogous problems

Figure 4.2: The number of participants who verbalized at least one search for analogous problems, by problem and experience (CS1 →light, CS2→dark

4.3.1 Problem Solving Process

Here I 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.

Reinterpreting the Problem Prompt

Reinterpretation is critical to understanding the nuances and ambiguities in problems [41]. 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 4.1, 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? 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 potentially disruptive task switches.

Searching for Analogous Problems

Programmers draw upon knowledge of previously encountered problems to provide insight into new problems [45, 105]. 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 4.2 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

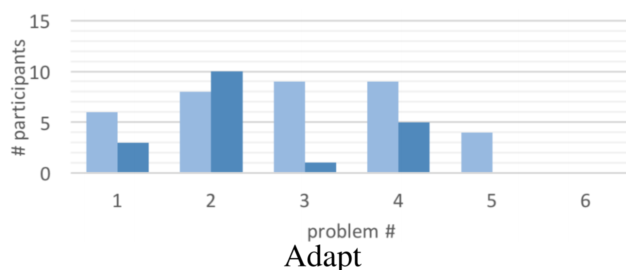


Figure 4.3: The number of participants who verbalized at least one solution adaptation, by problem and experience (CS1 →light, CS2→dark).

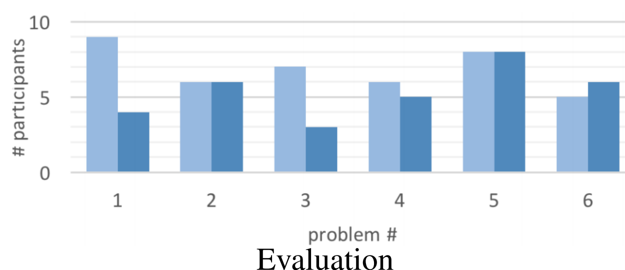


Figure 4.4: The number of participants who verbalized at least one solution evaluation, by problem and experience (CS1 →light, CS2→dark).

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 also 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.

Adapting Previous Solutions

Just as programmers rely on prior knowledge to conceptualize novel problems, they also rely on previous solutions [56]. 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.

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 4.3 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."

Evaluating Solutions

Evaluation of a solution, including analysis and testing, are critical to successfully solving programming problems [59]. 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 4.4 shows only about half of participants verbalized evaluating their solution. 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

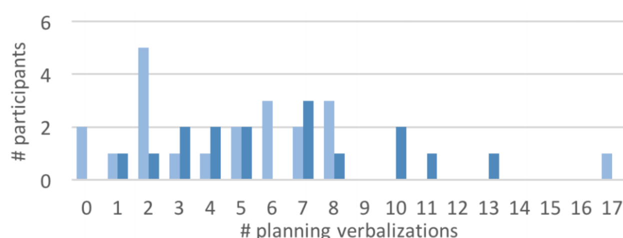


Figure 4.5: Frequency of participants' planning verbalizations across all problems, by experience (CS1 →light, CS2→dark).

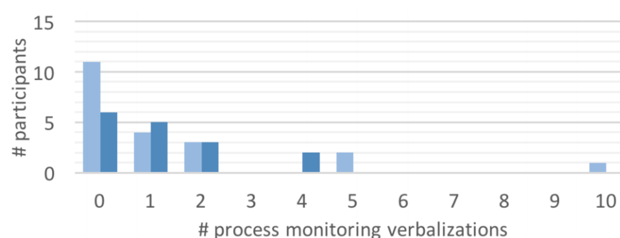


Figure 4.6: Frequency of participants' process monitoring verbalizations across all problems, by experience (CS1 →light, CS2→dark).

outcome was an increase in confidence allowing the participant to continue onto the next sub-problem or problem. For example, after evaluating, P10 (CS1) said: "All right. I am satisfied with this solution."

4.3.2 The Role of Self-Regulation

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

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 [105]. We expected that few participants would exhibit planning given their inexperience.

In fact, as you can see in Figure 4.5, the majority of participants did verbalize 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 which was not significantly different (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, "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.

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 4.6 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

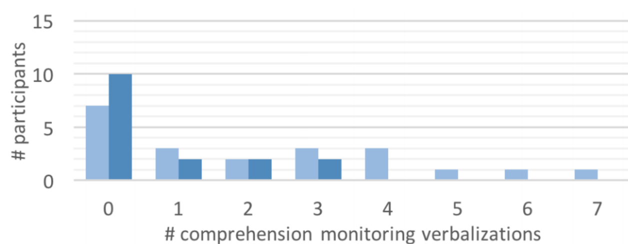


Figure 4.7: Frequency of participants' comprehension verbalizations across all problems, by experience (CS1 →light, CS2→dark).

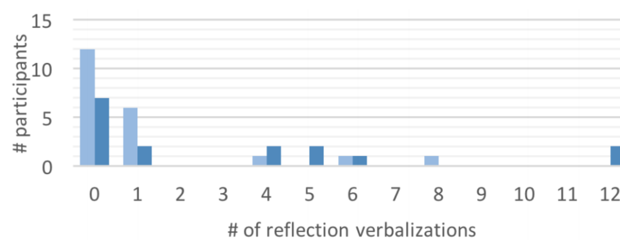


Figure 4.8: Frequency of participants' reflection verbalizations across all problems, by experience (CS1 →light, CS2→dark).

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.

Comprehension Monitoring

The programming self-regulation 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 4.7 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-read 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 difference in the types 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 (e.g. “I’m not sure what the length means?”, P11, CS1). When participants monitored problem comprehension, they indicated statements of confusion, as in P18’s (CS1) need for domain knowledge: “And so actually, I don’t know how this golf scoring works. How does the golf scoring work?” We found no differences in the content of CS1 and CS2 participants’ comprehension monitoring.

Reflection on cognition

Our framework argues 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.

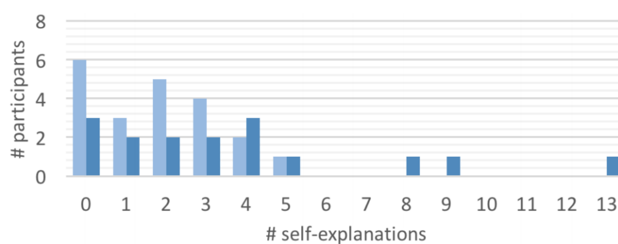


Figure 4.9: Frequency of participants' self-explanation verbalizations across all problems, by experience (CS1 →light, CS2→dark).

Reflection was more common than we expected. Figure 4.8 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 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."

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 4.9 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.”

4.3.3 Self-Regulation and Errors

Prior sections described 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 described in Section 4.2, 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 ($p=0.34$, $H=0.88$). However, on the most difficult problem (problem 5 in Table 4.2) the CS1 group did make 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 4.4 shows the resulting model for all participants. We found a significant model ($F(8,28) = 2.66$, $p=0.026$), with an $R2$ 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 4.4 shows the two resulting models for each group. The model was significant for CS1 ($F(7,13)=3.11$, $p<0.05$), with an $R2$ 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 ($p<0.05$)

Variable	All participants			CS1			CS2		
	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	1.5	0.09
# CS Courses	-1.49	0.9	-0.3	-	-	-	-	-	-
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	0.6	0.11
Comp. Monitoring	0.61	0.51	0.23	0.62	0.63	0.23	-2.9	0.99	**-1.09***
Reflection	0.2	0.32	0.13	1.1	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***
R2	0.43	0.62	0.66						
F	**2.66***	**3.11***	2.23						
*p < 0.05									

Table 4.4: Three models predicting errors from demographic and self-regulation variables. Unstandardized coefficients (B), standard errors (SE B), and standardized coefficients (β).

and ~1 error increase of for each verbalization of self-explanation ($p<0.05$).

4.4 Discussion

These 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 I 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 [96] and their infrequent comprehension monitoring and planning: they rarely reflected on their understanding of the problem or their code. Table 4.4 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 4.3.2). 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 were doing or why. That said, the more that CS2 participants engaged in planning and comprehension monitoring, the fewer errors they created (Table 4.4). 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 [11, 22, 105]. 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 [33] 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 for this finding 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. For instance, 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 [31, 59], and that it can be productively taught to novices [11], 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. These results of this study 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.

These results have several implications for teaching. If self-regulation behaviors are critical to programming success as prior work suggests [31, 59], it should be explicitly taught. Prior work has shown that self-explanation [11], and problem solving frameworks 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, and how to interleave it with syntax and semantics, what pedagogy to use). While only planning and comprehension monitoring had a negative relationship with errors in our study, other forms of self-regulation might also be taught explicitly. For instance, is instruction on

metacognitive reflection [32] 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, this study 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 [21, 57], 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 these results, we view these findings as an important first step in understanding the relationship between self-regulation and programming problem solving.

Chapter 5
NOVICES' IN-SITU SELF-REGULATION

Chapter 4 established that in a laboratory setting novices' do self-regulate but do so poorly. In this chapter I establish that the same also occurs in authentic programming settings.

The previous study investigated novices self-regulation providing a baseline expectation of novice programmers' self-regulation skills. However, a key limitation of this and other prior work is that it is all done retrospectively, asking students to reflect on their work *after* it has occurred. Therefore, little is known about how novices self-regulate while they program, and how this self-regulation might differ from what they recall about their self-regulation. In this chapter I present a study [1] that addresses this limitation, providing additional information about novices' self-regulation behaviors and skills by investigating novices' in situ self-regulation.

5.1 Introduction

There are many reasons to suspect that retrospective data and findings from previous work on self-regulation for computing might not generalize to in situ settings. In situ programming happens in a variety of environments (office, classroom, at home), where distractions might be more abundant. It happens in settings that are not time-regulated, often unfolding over many hours or days across multiple contexts. Similarly, programming may include teaching assistants, peers, and the Internet. All of these factors may be difficult to recall retrospectively, missing nuances about self-regulation that learners might engage in, but not remember.

[1] First published: Dastyni Loksa, Benjamin Xie, Harrison Kwik, and Amy J. Ko. Investigating Novices' In Situ Reflections on Their Programming Process. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, pages 149-155. ACM, 2020.

To investigate this generalizability gap in prior work, I studied novice students' programming self-regulation *in situ* across a 10-week series of four 2-week programming assignments, investigating the following research questions:

RQ1 When prompted to reflect on process in situ, what degree of in situ self-regulation do learners engage in?

RQ2 What challenges do students report encountering when attempting to reflect on their programming process in situ?

5.2 Method

To answer these research questions, we asked learners in a 10-week course to write in journals during programming sessions, reflecting on their problem solving and self-regulation. Using reflective journals is one of a few methods of measuring self-regulation [120] and have been used in CS to enhance programming skills [23]. For this study, journals served as both a prompt to self-regulate during problem solving and a record of the self-regulation the participants engaged in. Therefore, rather than just exposing how students work in the absence of being observed, our data reflects more of a best-case scenario for programming, where there is a scaffolded prompt to think about and write about their programming to support their problem solving.

5.2.1 Course and participants

We partnered with an instructor of one section of a required front-end web development course in the information science department of a large public research university. Participants consisted of all 31 undergraduate students enrolled in the course, of which all had passed at least one prerequisite programming course covering Java and basic data structures. Of the 31 students, 25 identified as men, 6 as women. One reported being in their 2nd year of undergraduate study, 13 in their 3rd year,

and 17 in their 4th year.

The course required students to complete 4 projects over 10 weeks. The first project required students to create a personal website using HTML and CSS. The second project required students to create a web-based game written in JavaScript. For this project, students selected the game they wanted to create and were given a variety of suggestions from classic arcade games like Pong or Breakout to casual mobile games like Threes or Bejeweled.

Programming Behaviors	Definition
Interpret prompt	Statements about or demonstrating interpreting or questioning the prompt, reconsidering actions in reference to the prompt or decomposing the problem into goals requirements and or sub-problems.
Search for analogous problems	Statements about or demonstrating intent to use code they have previously written or use of examples from outside sources.
Adapt a solution	Statements of or demonstrating changing or refining code.
Evaluate	Statements of or demonstrating testing or evaluating outcomes, intent to test a solution, or identifying why code was not meeting expectations.
Self-Regulation	**Definition**
Planning	Statements of intended work goals or requirements, or an intended order of work
Process Monitoring	Statements of start times, stop times, duration of coding session, about work being started identifying work currently in progress, when a task is complete, or statements that identify actions as part of their process.
Comprehension monitoring	Statements identifying known or unknown concepts or solutions.
Self-explanation	Statements of code explanation for increased understanding.
Reflection	Statements reflecting on prior thoughts of behaviors.
Rationale	Statements that provided rational to decisions or behaviors.

Table 5.1: Programming and self-regulation behaviors (adapted from Chapter 3) coded in the journals, with definitions.

The third project of the course required students to create a data explorer using the React framework [48] that allowed a user to interactively explore a data set, such as that exposed by a

public web API. What API to use, what data to present, and how to present that data was up to the student, but the project required that the app was both responsive and accessible (perceivable to screen readers). The fourth project for the course required students to create a messaging application using the React framework and a Firebase back-end. This project could be done individually, or in pairs, and required user accounts, authentication, and client-side routing to create a single-page application.

5.2.2 Data collection

To gather data about in situ self-regulation for RQ1, the instructor required students to submit a programming journal for each of their four projects. To ensure that the students had the language to describe their self-regulation in their journals, I taught students definitions of self-regulatory behaviors by giving a 20 minute lecture on the first day of class. This lecture introduced the problem solving framework (Section 3.1) and programming self-regulation framework (Section 3.2) and defined programming as a series of iterative problem solving behaviors drawn from prior work. For this study I refined the frameworks to include only the behaviors shown in Table 5.1 where the top is the problem solving behaviors and the bottom the self-regulation behaviors. The behaviors included were 1) reinterpreting the problem prompt, 2) searching for analogous solutions, 3) adapting previous solutions, 4) implementation, and 5) the evaluation of implemented solutions. All of the behaviors from the self-regulation framework included; 1) planning, 2) comprehension monitoring, 3) process monitoring, 4) self-explanation, and 5) reflecting on cognition. In the remainder of this chapter all behaviors in Table 5.1 will be referenced as self-regulation behaviors. We provided the definitions to these self-regulation behaviors and provided students with journaling instructions on the course website for later reference.

We instructed students to journal about the start and stop times of each coding session, their progress through the problem solving activities in Table 5.1, and to use the journal as a place to self-

regulate in the six different ways described in Table 5.1. To ensure some consistency in journaling and help students understand the expectations, we provided an example journal that demonstrated how a journal might cover all of the behaviors and demonstrate the level of detail we expected. Because the emotions, struggles, and successes of programming can be very personal, students were assured that their journals would be kept anonymous and we did not enforce a structure nor did we require journals contain specific content. This allowed students to authentically journal about their programming and encouraged including what they valued and expected to be useful. To provide additional scaffolding for reflecting on their process, we provided feedback on the first journal each student submitted, identifying where they could expand on the content, clarity, and depth of their journaling for future journals.

To understand the challenges students' encountered when trying to reflect on their programming process (RQ2), we required students to fill out a survey when submitting each of their assignments and corresponding journals. It asked two open response questions:

- *Your journal is to help you reflect on your process. Review your journal and briefly describe all points where you had trouble reflecting on your process, and/or writing parts of your journal.*

- *Think back to any time in the last two weeks where you stopped and reflected on your programming process when you were not programming or writing your journal and found it difficult. Describe why it was difficult.*

Our primary source of data was the student journals. Despite the journals being part of their grade for the course, some students did not submit journals. In total, we collected 106 journals with a combined total of 4,227 statements of students reflecting on their programming. Students' journals varied in level of detail, with some being quite extensive, as in this example entry: "*Initially I thought I was going to pseudo code a bunch of stuff, but instead I settled for repurposing a bunch*

Line - Codes	Journal Statement
...	...
A- *Evaluate*	Jumbotron inside a container seems too centered, made a manual "container"
B- *Reflection*	Project became really easy after I solidified my idea for the css layout and got it mostly done
C-	Friend suggested bootstrap navbar, that seems a lot easier than doing it with raw css
D- *Evaluate*	Bootstrap navbar was stacking everything on the right
E- *Search*	Took a while of looking through documentation, but apparently that was part of "mobile first" had to define what size it flattens at
F- *Evaluate*	Added pictures from my projects and a picture of my bird, think everything looks good
G-	Was using a stock icon, but I have a personal logo I use in other places, I guess it fits
H- *Evaluate*	Tried setting footer bottom to absolute 0, did not work, will just not mess with it
I- *Reflection*	Decided to add a gradient to make it look nicer, was very surprised how easy it was and how much it improved the look
J- *Interpret*, *Reflection*, *Evaluate*	Was looking through spec, realized I forgot highlights, they didn't work for a bit until I made them really specific

Table 5.2: The second half of one students' journal for the first homework, showing the codes applied from Table 5.1. This participant was in the *high* coverage cluster.

of code from a previous exercise that takes user input and posts messages (Chirper from a previous exercise.)" Others, in contrast, were quite terse: "*changed how I had BrowserRouter set up.*"

5.3 Results

5.3.1 RQ1: What degree of in situ self-regulation do learners engage in?

To answer this question, we performed three analyses:

- We coded and computed the frequency of the behaviors in Table 5.1.
- We investigated whether there were distinct patterns of student behavior through clustering.
- We analyzed students survey responses about their journaling process for the “maturity” of reflection.

Frequency of behaviors

To understand the frequency of self-reflective behaviors, we coded each statement of each journal entry to identify if it demonstrated one or more of the self-regulation behaviors detailed in Table 5.1. To ensure the codes we applied were well-defined and consistent, we iteratively refined the code definition by having two authors apply the codes to a set of 430 (10% of the total 4,227 statements) randomly selected journal statements, using adjacent journal statements for context if necessary. To drive refinements, we discussed disagreements, refining definitions, and coded a new set of randomly selected journal statements. After four rounds of iteration, the authors reached 83% agreement on this sample data set. I then coded all remaining statements. Table 5.1 shows the final code definitions for each of the self-regulation behaviors. Table 5.2 shows an excerpt of one student’s journal, showing a session of writing, testing, and refining some CSS.

Based on these codes, we computed the frequency of each type of code in each student journal. Table 5.3 (Left) shows the range of the number of codes found in journals for each behavior. The

Behavior	#1	#2	#3	#4
Interpret	0-1	0-1	0-1	0-1
Search	0-7	0-2	0-2	0-3
Adapt	0-6	0-4	0-1	0-1
Evaluate	0-11	0-6	0-8	0-14
Planning	0-19	0-9	0-19	0-7
Process Monitoring	0-25	0-25	0-54	0-43
Comprehension monitoring	0-6	0-2	0-4	0-2
Self-explanation	0-24	0-17	0-13	0-9
Reflection	0-13	0-7	0-7	0-8
Rationale	0-17	0-8	0-8	0-7

Code	High (12)	Moderate (12)	Low (7)	Total
Process	100%	100%	100%	100%
Evaluate	100%	100%	100%	100%
Search	100%	100%	86%	97%
Reflection	100%	100%	43%	87%
Explanation	100%	100%	29%	84%
Rationale	92%	100%	29%	81%
Planning	92%	75%	29%	71%
Comprehension	100%	33%	29%	58%
Interpret	75%	8%	0%	32%
Adapt	75%	8%	0%	32%

Table 5.3: **Left:** The range of student entries exhibiting each self-regulation or programming behaviors, by assignment, showing higher frequencies of evaluation, planning, process monitoring, and self-explanation than other behaviors and interpret only occurring once per assignment. **Right:** The three clusters of behavior (and size of cluster). Each percentage indicates the proportion of students in the cluster who exhibited the behavior at least once in a journal.

"Total" column in Table 5.3 (Right) lists the total percentage of students who journaled about each behavior at least once across all four journals. These two tables show that most participants (>80%) journaled about the following behaviors at least once across their four journals:

- *Process monitoring* (e.g. "Day Two - Start Time - 1:30 PM | End Time 8 PM")
- *Evaluating solutions* (see Table 5.2.A,D,F,H for examples)
- *Searching for analogous problems* (see Table 5.2.E for example)
- *Reflecting on their cognition* (see Table 5.2.B,I for examples)
- *Self-explanations* (e.g. "*ALL THAT WAS WRONG WAS THAT I DIDN'T LINK IN THE RIGHT VERSION OF JQUERY ASDFGHJKL;[sic]*")
- *Rationale* (e.g. "*I think before I do that I should make the page look prettier and better organized, because I am reusing a lot of code for API requests*")

The least common behaviors included:

- *Interpreting* the prompt (see Table 5.2.J for example)
- *Adapting* previous solutions (e.g. "*The majority of this chat application will come from exercise sets so I'm taking code from those assignments and picking the components that apply to the project*")

Clusters of behaviors

The study in Chapter 4 suggests that there is large variation in the self-regulation behaviors of novices. To better understand this variation, we attempted to cluster students based on which behaviors they did and did not exhibit in their journals. We began by computing a binary variable for each student and each behavior in Table 5.1, true if at least one journal exhibited that behavior, and false otherwise. Then, we performed a visual inspection of this binary data and observed that there were potentially 3 patterns of self-regulation behavior. To verify this interpretation, we applied the K-modes unsupervised clustering algorithm [47] to the student data, using the binary variables as features, specifying K=3 to separate the students into 3 clusters.

The resulting clusters, shown in Table 5.3(Right), aligned with our visual inspection. One cluster, which we will refer to as the *high coverage* cluster, contained 12 participants whose journals had exhibited at least 9 of 10 of the behaviors in Table 5.1. These high coverage students typically were missing entries exhibiting the behaviors (*Interpret* the prompt and *Adapting* solutions). The journal excerpt shown in Table 5.2 is from a student in the high coverage cluster and shows evaluating, reflecting, and interpreting about a series of CSS problems. The second cluster, which we refer to as the *moderate coverage* cluster, had 12 participants who journaled about the most common behaviors (*Process* monitoring, *Evaluating* solutions, *Searching* for analogous problems, *Reflecting* on their cognition, creating *Self-explanations*, *Rationale*), but not the least common behaviors. The final cluster, which we refer to as the *low coverage* cluster, had 7 participants who journaled about the fewest behaviors, with *Process* monitoring and *Evaluating* solutions being the only behaviors all

participants in this group journaled about.

Maturity of reflection

Whereas our first two analyses considered frequency of reflection and patterns in reflection activity, our third analysis of the journals considered the "maturity" of the reflection itself. We defined maturity as the degree to which self-regulation was an integrated part of students' programming process. To analyze maturity, we qualitatively coded the responses that students gave to the two journaling survey questions, analyzing how students wrote about their journaling process. Two researchers inductively coded the responses identifying varying categories of detail in participant reflections on their process, developed a coding scheme based on those categories, and used that scheme to independently code the survey questions, following the approach from prior work [109]. The researchers reached 88% agreement before reviewing and reconciling discrepancies, coming to 100% agreement on all assigned codes.

The final coding scheme consisted of three codes representing the level of reflection maturity in participant responses. Participants who demonstrated *mature* reflection were those who identified that they were struggling with reflecting while programming because the reflection conflicted with their process. For example, one participant with *mature* reflection stated, "*When I hit really difficult bugs, I don't want to reflect on them or journal, I just want to look at my code and chase them down.*" These participants demonstrated a high awareness of their current process and how reflection interfered with it. Perhaps unsurprisingly, due to asking students to incorporate a new skill into their programming process, there were no students in the *mature* category that demonstrated high awareness of their process without identifying that it interfered with their current process. They often stated that they opted not to engage in journaling during programming, choosing to journal after-the-fact to meet the journaling requirement. Another set of participants we identified was those who were actively *integrating* reflection into their process. This group provided indications that they

were still developing a programming process, often reporting struggling because the act of reflecting was difficult rather than because it conflicted with any current process. For instance, one *integrating* participant expressed, "*It is [difficult] because that I might not remember all the details all the time.*" The final set of participants we identified as being *process-unaware*. These participants stated that they did not reflect on their process, that they had no difficulties reflecting while providing no additional details, or responded to the question by describing their process or a segment of code that was difficult rather than an aspect of their process. For example, one *process-unaware* participant responded, "*I never really stopped and thought about something being hard, I just started looking through google/documentation.*"

5.3.2 Implications and future work

These results have important implications for future self-regulation interventions and research. These results suggest that educators seeking to scaffold the development of self-regulation skills should strongly consider the robustness of students' current self-regulation skills. The challenges reported by our participants demonstrate that interventions intended to help build self-regulation skills may act to needlessly slow down and hinder students' ability to be productive. Alternatively, students that would fall into the *high coverage* or *moderate coverage* clusters may disregard the intervention in favor of their current workflows resulting in wasted efforts with no benefits. Additionally, without careful training and scaffolded practice *low coverage* students may struggle to begin to develop necessary self-regulation skills at all, remaining in a state of *low coverage*. Instead, educators may want to have tiered, faded scaffolding, systems that first carefully train *low coverage* students in self-regulation skills and the ability to reflect on them, helping them to achieve *moderate awareness*. Additionally, educators might want to take into account the self-regulation behaviors that students are more and less apt to engage in. Some work is already attempting to emphasize the importance of developing explicit practice *Interpreting the prompt* from the beginning [91] and similar efforts

might be needed for *Adapting* to help students be more aware of their process and help them more quickly achieve *high coverage* levels of self-regulation.

We believe further research into understanding the development of novice programmers' self-regulation is necessary. First, future work should replicate our findings in other authentic programming settings, using other programming languages, and in other cultures. Future work should also refine the training and instruments used in our study to more accurately measure self-regulation in situ. Researchers should leverage our awareness cluster findings as a basis for further investigations on the development of self-regulation skills in programming. Future work should also explore the order in which novices develop particular self-regulation behaviors. Similarly, future work should investigate connections between categories of reflection and the clusters of behavior coverage.

Despite the limitations on the validity and generalizability of our results, our findings are an important step in understanding the in situ self-regulation of novice programmers. With further research, improved instruments, and refined theories, we hope for a future where educators understand self-regulation development and leverage that understanding to support the development of robust self-regulation skills for all students.

Chapter 6
EXPLICITLY TEACHING AND SCAFFOLDING METACOGNITION

The two previous chapters investigated novices' self-regulation in programming, primarily for the purpose of being better prepared to explicitly teach novices the mental skills of programming. This chapter follows from those results, contributing an approach to promote metacognitive awareness in introductory programming settings. This chapter addresses the thesis statement by describing a study[1] investigating the approach's effect on the productivity, independence, self-efficacy, and growth mindset of novices.

6.1 Introduction

The results discussed in the previous two chapters show a trend of novices having difficulty being aware of their own thinking. Chapter 4 found that when novices did self-regulate, it was infrequent and shallow. In Chapter 5 we found that, in an authentic programming context, novices found it difficult to even reflect on their thinking and process, finding it too abstract or that it conflicted with their own process. These results suggest that one way to support novices might be to first explicitly teach, and help learners develop, *metacognitive awareness*. Given that programming requires disciplined thinking and self-awareness [83], the results of the previous chapters highlight that computer science education may be failing learners by assuming that students are aware of and have developed, or can independently develop, these invisible skills that are required to succeed.

In the following sections of this chapter I present, and describe an evaluation of, an integrated set

[1]First published: Dastyni Loksa, Amy J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: effects of explicit guidance. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, pages 1449-1461. ACM, 2016.

of four interventions to explicitly teach and support the development of metacognitive awareness.

6.2 Interventions

These four interventions are designed to teach learners how to converge toward programming solutions while incentivizing them to recognize, evaluate, and refine their problem solving strategies:

- Provide explicit instruction on the goals and activities involved in programming problem solving. Frame problem solving in programming as a set of distinct stages (which we describe shortly).

- Prompt learners to describe their problem solving state. When learners ask for help from a person, or from software such as an intelligent tutoring system or learning technology, prompt learners to describe the problem solving stage in which they are engaged. This encourages additional reflection on their problem solving.

- Provide a physical representation of problem solving stages to help learners monitor their state. Provide a physical handout that details the programming problem solving stages and encourages learners to track which stage they are in, peripherally prompting learners to be aware of what actions might be appropriate next.

- Provide context-sensitive problem solving prompts. To reinforce metacognitive awareness during code editing, offer contextual hints that prompt learners to recognize the problem solving stage they are engaged in.

Utilizing the problem solving framework from Section 3.1, one can instantiate instruction on the problem solving behaviors and the three proposed forms of metacognitive prompts (a handout modeling the problem solving stages, help request prompts, and context-sensitive help) in many different ways. For example, in online learning technologies, these interventions might be built into

automated tutorials or online IDEs. In classrooms, they might be interactive activities, lectures, TA prompts, or even grading policies.

Our instantiation of the explicit instruction was a 1-hour problem solving lecture teaching the six programming problem solving behaviors as described in section 3.1. The instructor began the lecture with a physical book sorting exercise. He asked the campers how to sort the books by size and followed their verbal instructions. Next, he conducted a guided-discovery discussion, asking the campers how they knew how to sort the books in that way and why they sorted the books that way. The campers discussed the how and why amongst themselves until they reported that they understood the problem. The instructor then prompted for more explanation until it became apparent to campers that the questions were not as simple as they initially seemed. The instructor used this realization to trigger a discussion of each of the six problem solving behaviors, starting with reinterpreting the problem prompt. Campers tried to identify the next behavior in the process as a group at the instructor's request. Once the campers identified the next behavior (or the instructor identified it when campers ran out of ideas), he tied abstract concept of the behaviors to the concrete book sorting problem the lecture began with. The following section describes how we evaluated our instantiation of the interventions and provides one example of how they might work together in practice.

6.3 Method

To explore and evaluate the impact of these metacognitive interventions, we conducted a controlled experiment across two 2-week camps with 48 high school students who signed up to learn basic web development. We hypothesized that our interventions would improve learners' ability to describe their problem solving progress, strengthen their self-efficacy (their degree of confidence in their ability to carry out a task [6, 7]), foster growth mindsets (their theories about whether ability is learned or innate [30]), and ultimately produce a higher quantity of functional code. The goal of

our experiment was to compare a traditional version of a web development camp (our control) with an experimental version of the same camp that included the four interventions. In the rest of this section we describe our camp, data collection, and the results of our investigation into our predictions. We then discuss the implications of our findings for the broad landscape of efforts to teach coding in classrooms, tools, and online.

Participants

Our participants were campers in a university-sponsored summer youth learning program. The program was based in a region with a large software industry, so many of the campers likely knew someone with coding skills. Campers in the youth program have historically been from upper-middle class families with college-educated parents, and have typically been only 20-30% female. Campers and parents were not aware of any difference between the two camps other than their scheduled time. The youth program managed registrations, recruiting 25 campers in the experimental group and 23 in the control. From this point forward, we refer to campers with a letter indicating their group followed a unique number (e.g. E27 is an experimental camper and C75 a control).

The experimental group included 8 females and 17 males. Two campers listed English as a second language. The control group included 8 females and 15 males, and all listed English as their primary language. The two groups were largely indistinguishable: they did not miss class at different rates (Kruskal-Wallis, $H=2.2$, $p=0.138$), they had similar grade levels ($X2=4.1829$, $df=3$, $p=0.242$), and similar self-reported programming and web development experience ($X2 = 2.669$, $df=1$, $p=0.102$).

The Camps

Each camp consisted of ten 3-hour weekday sessions from 9am to 12pm (experimental) and from 1pm to 4pm (control). We placed the experimental group in the morning to bias any instructional

Day 1	HTML lecture and activity
Day 2	1-hour problem solving lecture (experimental only) Problem solving stages handout and prompts (experimental only) CSS lecture and activity; 1-hour additional CSS activity (control only)
Day 3	JavaScript lecture and activity Growth mindset development exercise
Day 4	React lecture and Interactive activity Growth mindset development exercise Problem solving reminder (experimental only)
Days 5-9	Free development time
Day 10	Project presentations

Table 6.1: The camp schedule, with experimental camp's additions as noted.

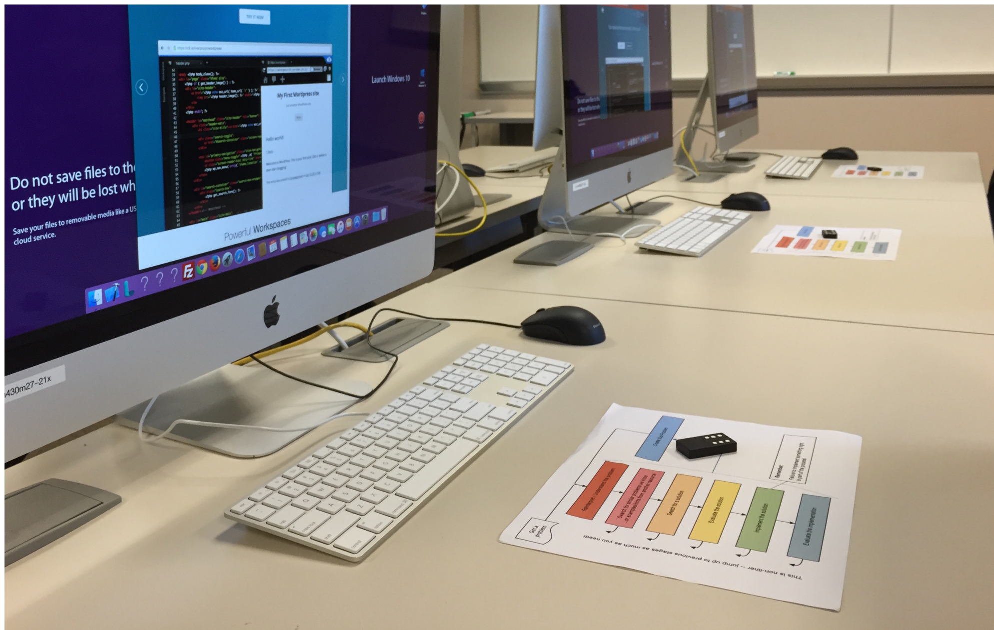


Figure 6.1: The paper handout and physical token we gave to campers to track their problem solving stage.

improvements toward the control group (though this may have introduced other confounds, as we discuss later). Both camps took place in the same university computer lab. Campers worked in the Chrome web browser and Cloud9, a web-based IDE (http://aws.amazon.com/cloud9/).

The Instruction

We aimed to teach concepts, syntax, and semantics of HTML, CSS, and JavaScript with a focus on the React JavaScript framework (facebook.github.io/react). Our goal was for campers to feel capable of learning more about these technologies, but not necessarily capable of developing interactive web sites with them independently. We chose the React framework because it is based on a powerful but highly constrained view abstraction, which meant that there are only a small number of ways to implement any particular functionality. This made measuring task completion more straightforward, as we describe later in our results.

As Table 6.1 shows, the camp included 4 days of lectures and practice, followed by 5 days of self-directed programming time on a course project. The lead instructor presented HTML, JavaScript, and React lectures to both groups. Another instructor presented a CSS lecture and a growth mindset exercise to both groups. Three additional undergrads also acted as helpers to the campers. All members of the instructional team had at least novice experience with web development. The lead instructor had experience as a professional software developer but had no experience running camps or teaching programming.

The experimental group began with the 1-hour lecture and book sorting problem described in Section 6.2. After the lecture, we provided the experimental group with a physical handout of the problem solving behaviors (shown in Figure 6.1) and a physical token so they could track their current state on the handout (the second part of our intervention). We instructed campers to track their progress through the behaviors as they worked on their website and to reflect on and adjust their strategies. While the problem solving lecture detailed what programmers must achieve in the six behaviors, it did not prescribe how they achieve it. We did not mention any particular strategies or resources to use for each behavior. The one exception to this is a mention of the development of sub-problems, which the instructor mentioned in the lecture and noted in the handout. The instructor encouraged the use of the Idea Garden, which mentions strategies such as working backwards.

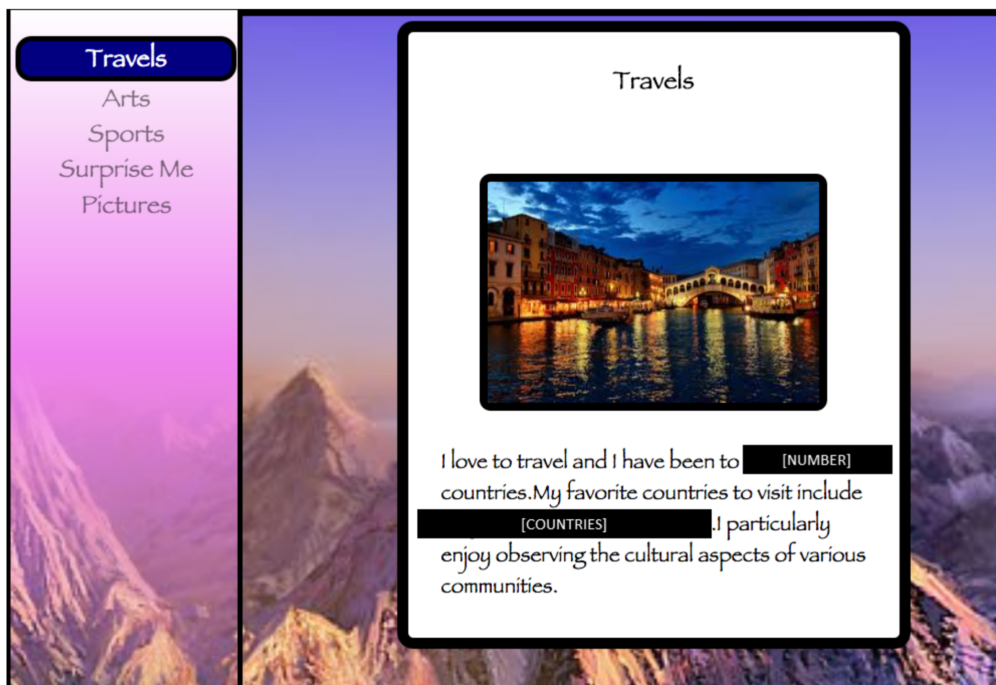


Figure 6.2: Camper E27's final project, showing buttons that link to different interests (left) and content and images (center). Details have been anonymized.

The Project

After the four days of lecture and practice, campers in both groups spent the remaining five work days on a class project. The project was to build an interactive, React-based single-page web application that contained both static and interactive content about campers' interests. Figure 6.2 shows an example of a camper's final site. To scaffold the project, we provided a basic architecture for the application. We then provided a set of 20 progressively more difficult tasks for campers to complete at their own pace (see Table 6.2).

Task	Content	HTML	CSS	JS
Add a window title to the web page		✓		
Create objects to represent each of your interests	✓			✓
Change the background color and add a border to your page			✓	
Create a space for each of your interest's names		✓		✓
Add a component that displays a photo of your interest				✓
Display interest text paragraphs in their own <div>tags				✓
Give your page a background image			✓	
Give the content area a background color and rounded border			✓	
Use a component to display a page title stored in a variable				✓
Give each paragraph a unique style using .map()			✓	✓
Make a "Surprise Me" button that shows a random interest	✓			✓
Style your buttons with a border and transitions			✓	
Create a menu component with two buttons			✓	✓
Make the menu navigate between the interests and "about me" pages				✓
Fill your "about me" page with content about you	✓	✓		✓
Make the title match the currently selected page				✓
Add an image to "about me" page that changes when clicked				✓
Embed a video in your interest's content area		✓		
Link your images to an external page	✓			✓
Create a photo gallery that displays six images	✓	✓	✓	✓

Table 6.2: Condensed versions of the prescribed tasks given to the campers and the skills that each task required.

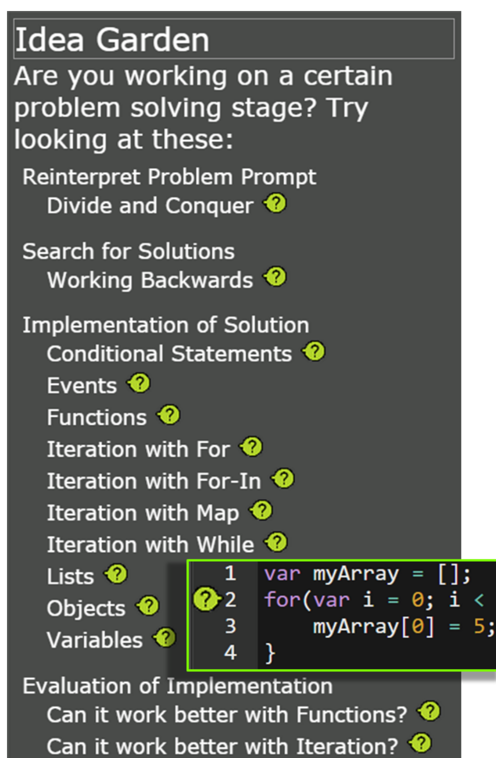


Figure 6.3: (Main) The Idea Garden panel in the Cloud9 IDE as campers see it when they opened the panel for the first time. (Callout) An example of the Idea Garden decorating the code with an icon. Here, the icon links to the Iteration with For hint.

During both the after-lecture activities (during the first week) and project work time (during the second week), campers in both groups had access to several types of help. We gave campers access to the PDFs of the lectures along with HTML, CSS, and JavaScript “cheat sheets.” We also encouraged campers to find online resources on their own. The two instructors and three helpers also offered help upon request. The helpers’ goals were twofold: 1) to get the camper on a more productive path without giving them a solution and 2) to gather data about the camper’s metacognitive awareness and problem solving strategies. To achieve these goals, helpers provided assistance only when asked for by the students, and they never provided code.

When responding to a camper’s help request, helpers first asked the camper two questions: 1) “Describe the problem in as much detail as you can” and 2) “What have you tried so far?” Additionally, helpers asked the experimental group, “What problem solving behavior do you think you are in? (The third of our interventions). After these questions, the helpers provided assistance. Next, the helpers recorded detailed observations about the problem(s) the camper had encountered and the assistance provided. At the end of each day, helpers transcribed their notes, elaborating on details they did not capture previously.

To provide context-sensitive problem solving prompts to the experimental group (the fourth

of our interventions), we implemented the Idea Garden [17–20, 50] in a panel of the Cloud9 IDE (see Figure 6.3, main). The Idea Garden, as a design concept, entices programmers to consider new ideas when they are stuck on a task. In this manifestation, we reinforced the problem solving behaviors by housing the Idea Garden's 14 hints under headers corresponding to the six behaviors. When campers triggered a programming "anti-pattern", such as forgetting to use the iterator in a for loop, the Idea Garden placed an icon on the screen next to the problematic line of code (Figure 6.3, callout). If the camper then clicked on the icon, the titles of hints relevant to the problem became highlighted.

Data Collection

At the end of each camp day, campers completed an end-of-day survey. To learn about the campers' metacognitive awareness during the camps, we adapted the techniques of [108, 113], asking campers to reflect on a difficult task and respond to the survey question "How did you solve this problem? If you didn't solve it, what did you try?"

To measure campers' programming self-efficacy, we adapted the scale by Askar et al. [4] to reflect web development tasks. The eight survey prompts were on a 5-point Likert scale and featured statements such as "I can write syntactically correct JavaScript statements", "I can complete a programming project even if I only have the documentation for help.", and "When I get stuck I can find ways of overcoming the problem."

To measure campers' growth mindset, we used previous programming aptitude mindset measures of Scott & Ghinea [102]. The three survey prompts were also on a 5-point Likert scale and included the statements "I do not think I can really change my aptitude for programming.", "I have a fixed level of programming aptitude, and not much can be done to change it.", and "I can learn new things about software development, but I cannot change my basic aptitude for programming."

To measure productivity, helpers saved the campers' source code at the end of each camp session.

We also captured the experimental group's use of the Idea Garden, modifying a Cloud9 event logging mechanism to report Idea Garden interactions like opening a hint. The experimental group's end-of-day surveys included three questions about how campers used the Idea Garden as a resource.

6.4 Results

Because the camp was an experiment, everything we described in the previous section was identical for both groups, with the exception of the four things added to the experimental group: 1) the problem solving lecture, 2) the handout in Figure 6.1, 3) the help request prompts, and 4) the Idea Garden help shown in Figure 6.3. This section presents the effects of these interventions, beginning with a qualitative description of the campers' experiences and outcomes to give context to our results. I then discuss the effects of our intervention on metacognitive awareness, help requests, productivity, self-efficacy, and growth mindset. All statistical hypothesis tests we report were non-parametric Kruskal-Wallis or Chi-squared tests. All references to students in the camp are formatted by the condition, E for experimental and C for control, followed by an anonymized ID number.

6.4.1 Camper Experiences

As with any learning environment, the campers had a diversity of skill, engagement, and performance. Some campers relied heavily on the physical handout, while others only referenced it when prompted by camp helpers. Some of the most productive campers created their own tasks and used all the tools at their disposal to accomplish those tasks. For example, camper E40 (a 12th grade male) asked for the most help and earned the second highest productivity score. He discussed his problem solving activities and interacted frequently with the Idea Garden. On day 3, he read the iteration hints about for, for-in, and map and later asked for help iterating over his list of photos. On day 5 he said that the Idea Garden gave him new tactics: "yeah, it told me to try using a map function or a for-in loop and im [sic] trying to get them to work." On day 6, helpers observed him successfully

using iteration without help.

The control group also contained highly productive campers, but they appeared to be less independent. Campers C91 (10th grade male) and C92 (11th grade male) earned the two highest productivity scores in the control group, working together. C91 said, “Tell me what’s wrong here because I’m not going to bother figuring out what’s going on,” showing how quickly he gave up on solving problems independently. When C91 and C92 struggled they compensated by working together and repeatedly asking for help.

Other campers were less productive. For example, camper E50 (a 9th grade male) focused primarily on content changes and the most challenging task (the photo gallery) in Table 6.2, but did little work on any other task. He worked independently and tried to use the Idea Garden, but reported: “I tried looking at [the hint about using the JavaScript map function] and it wasn’t really useful”. He encountered many early stage learning barriers (described later) as well, saying things like “I don’t know where to start. I did display a photo, but I don’t know how to create a component.” C87 (an 11th grade male) also earned low productivity scores due to avoiding tasks requiring JavaScript and only requested help with CSS and HTML.

6.4.2 Impact on Metacognitive Awareness

The stories in the previous section suggest several differences between the groups. One difference we predicted was that our problem solving instruction would help campers be more aware of the strategies they used, enabling them to better identify and describe them.

To investigate this hypothesis, we evaluated metacognitive awareness by analyzing each of the responses to the end-of-day survey question “How did you solve this problem? If you didn’t solve it, what did you try?” The most salient difference in the responses was the presence or absence of specific problem solving strategies or tactics. For example, many campers wrote in detail about their efforts to solve a problem, such as “I did not solve the question. I googled it, and tried several

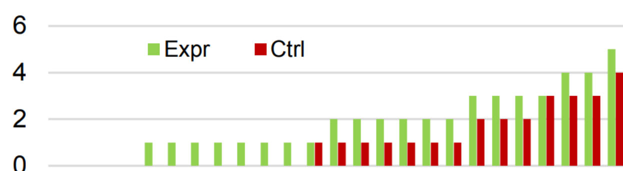


Figure 6.4: The total number of strategies mentioned in end-of-day survey responses by campers in each group, sorted by frequency. The experimental group mentioned more strategies than the control group.

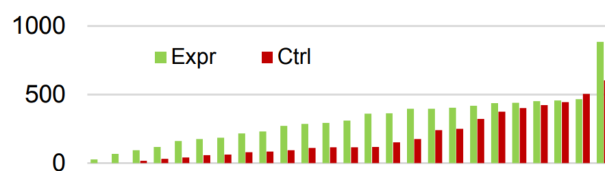


Figure 6.5: The total word count of all end-of-day survey responses by campers in each group, sorted by count. The experimental group wrote more than the control group.

bits of code, but I must have used them incorrectly, because they did not work." (C76) and "I looked at the slides and copied similar code just in the context of my code. But there was a small error between 'item' and 'items' which took a long time to figure out." (C84). Others were quite terse and simply mentioned asking for help, as in "teacher help" (C82) or "I asked an instructor." (C83). Some additional strategies mentioned included asking peers for help, searching google, copying and modifying previous code, and mental simulation of the code looking for errors. Two researchers counted the number of end-of-day responses per camper that described a specific strategy or tactic other than asking an instructor for help (reaching 90% agreement). After comparing these counts, we found that campers in the experimental group were significantly more likely to write an explicit description of a problem solving strategy (H=4.554, p=0.032) (see Figure 6.4). As shown in Figure 6.5, the experimental group campers also wrote significantly more words in their responses (H=6.326, p=0.011).

6.4.3 Impact on Types of Help Requested

Our instruction aimed to help campers be more aware of their current problem solving state, and therefore more capable of evaluating their strategies. Therefore, we predicted that the experimental group would be more independent and make more progress before requiring help than the control

Barrier	Definition from [56]	Representative Quote from Camper	Control	Experimental
Design	Did not know how to approach solving a problem.	“I’m incredibly lost. I think I’m on task 4?” – camper C92	9%	6.7%
Selection	Had an approach but did not know what language or API features to use.	“How can I get the title a different color?” – camper C95	27.8%	21.3%
Use	Had a language or API feature but did not know how to use it.	“I’m kind of confused on how to write an if statement to display the pictures...if the tab is PhotoGallery” – camper E42	34.4%	37.3%
Coordination	Did not know how to use two or more language or API features together.	“This is no longer working. They were separately but I tried combining them and it doesn’t” – camper C89	4.2%	3.2%
Understandin(	Observed a failure and did not have guesses about why.	“I added this photo code to my webpage and now my buttons don’t work” – camper E37	23.8%	28.8%
Information	Had a guess about why a failure occurred but could not get information to confirm	“I’m using getElementByID here in the HTML but it keeps evaluating to this ‘else’ so I know it’s not working” – camper E50	0.8%	2.7%

Table 6.3: Each row defines the barrier and gives an example from a help request, along with the percent of each type of barrier reported by each condition in their help requests. Highlighted cells are the higher of the two proportions.

group. For example, if a camper in the implementing a solution stage struggled with getting some JavaScript to work, exposure to the paper handout, the help request prompts, and the Idea Garden might remind them to search for an alternative solution, think of other similar problems they had solved before, or re-evaluate their understanding of the problem.

To detect this possible change in help requests, we classified the notes on each help request using a previously reported coding scheme on programming learning barriers [56]. We list the six barriers in Table 6.3, showing examples from campers. Each barrier is a general type of impasse that

Problem Solving Stage	Potential Barrier(s) Encountered
Reinterpret problem prompt	Design
Search for analogous problems	Selection
Search for solutions	Selection
Evaluate solution supposition	Selection
Implementing a solution	Use, Coordination
Evaluate implemented solution	Understanding, Information

Table 6.4: The barriers from [56] that might be encountered in a particular problem solving behavior.

learners typically encounter in programming tasks. Table 6.4 lists some of the barriers that might occur during particular problem solving behaviors. Two researchers coded the helper observations from camper help requests. They reached 88.75% agreement on 20% of the data and then coded the rest separately.

The helper to camper ratio (1:5) in each camp constrained the amount of requests (289 requests in the control, and 309 in the experimental), so we focused on analyzing the relative proportion of different types of requests. As shown in the two rightmost columns of Table 6.3, the proportion of help request types varied significantly by condition (X2=11.087, df=5, p=0.049). Campers in the control group requested assistance with design and selection barriers more often (devising a solution to a problem and identifying programming language and API constructs to implement it). In contrast, the experimental group requested more help with understanding and information barriers (how to debug their implementations). Though the difference in proportions of help request types was not large, campers in the experimental group were more likely to select a solution and implement it independently, allowing them to progress to evaluation before seeking help.

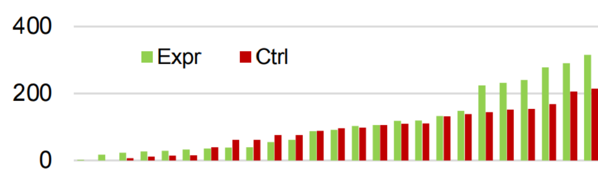


Figure 6.6: Campers' prescribed task productivity scores by condition, sorted in increasing order. The experimental campers' productivities were typically about equivalent to or higher than the control campers'.

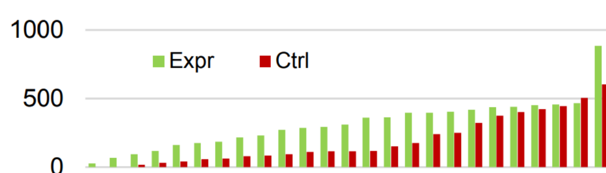


Figure 6.7: Campers' self-initiated task productivity scores by condition, sorted in increasing order. Experimental campers' productivities were significantly higher than control campers'. Values of zero are not visible.

6.4.4 Impact on Productivity

If our problem solving instruction was effective, we would also expect to see the experimental group finish more work than the control group. To test this hypothesis, we considered the degree to which campers completed prescribed tasks and self-initiated tasks for their project.

To measure these two kinds of productivity we counted the number of tasks completed, weighted by the category of tasks identified in Table 6.2 to determine a productivity score. Two researchers inspected each camper's final project source code and web site, checking which tasks they had completed. We only counted a camper's code as completing a task if it resulted in visible features on their website. React restricted the number of ways a camper could accomplish a task, making this assessment straightforward. For the self-initiated tasks, the same two researchers checked each camper's website for additional functionality, recording a description of its behavior and the code required to implement it. Campers in both conditions completed several impressive additions to their project, such as additional menu items in their profile page, widgets that displayed the current time, embedded videos, and a two-player "tic tac toe" game.

Tasks (prescribed or otherwise) required different amounts of work and thus had different levels of difficulty. Some tasks were simple content changes, while others required substantial JavaScript implementations. To account for this varying work in each completed task, we categorized tasks

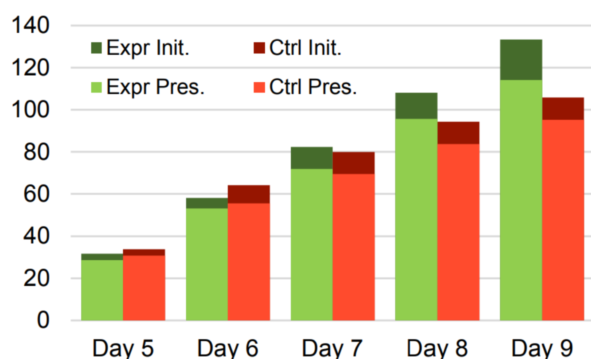


Figure 6.8: Cumulative average productivities per project day on both prescribed (light hues) and self-initiated (dark hues) tasks. The experimental group was increasingly more productive than the control group

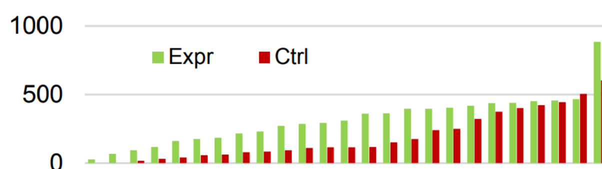


Figure 6.9: The campers' median lines of code changes per project day by condition. Experimental campers' amount of code changed was not significantly different from control campers' except on day 8.

according to which skills they required (as indicated in Table 6.2). Content tasks that only involved writing natural language, but not modifying markup or code, received 1 point. HTML tasks that involved adding or editing tags or HTML attributes received 2 points, as we considered these changes more difficult than modifying content because of the knowledge of markup syntax required. CSS tasks involved creating new CSS rules that interacted with HTML received 4 points, since they involved complex interactions with the DOM. Finally, JavaScript tasks that involved interactions with content, HTML, and CSS, received 8 points, as they required the most effort to complete and did the most to further campers toward the goal of developing a highly interactive website.

Comparing each group's weighted task completion scores revealed several interesting trends. First, as shown in Figure 6.6, the two groups completed similar amounts of prescribed task work in the same amount of time (H=0.0009, p=0.975). However, the experimental group completed substantially more self-initiated tasks: only 4 control group campers (17%) added additional functionality, compared to 11 experimental group campers (44%). This additional work led to the experimental group achieving significantly higher work scores (H=4.509, p=0.033), completing

over twice as much self-initiated work on average (as shown in Figure 6.7). Figure 6.8 shows that the experimental group's productivity on both prescribed and self-initiated tasks outpaced that of the control over time.

When we counted the lines of code that campers changed on each day of project work, there was no significant difference between groups (with the exception of day 8) (see Figure 6.9). This suggests that the experimental group got more work done with a comparable amount of code editing.

One potential confound in these results is the extent to which campers sought help: if the experimental group relied more heavily on the instructor and helpers, it may have explained their higher productivity. To investigate this, we checked the correlations between campers' help requests and total productivity scores, and found the opposite: the experimental group showed no significant association between help requests and productivity (Pearson: r(23)=0.278, p=0.179), whereas the control group did have a significant association (Pearson: r(21)=0.467, p=0.025). This suggests that the control group not only accomplished less work, but relied more on the helpers to complete this work.

6.4.5 Impact on Self-Efficacy

With the experimental group's greater productivity, we also expected to see a relative increase in self-efficacy when compared to the control group. To test this prediction, we calculated the mean of each camper's eight self-efficacy survey responses at the beginning and end of the camp, resulting in a score from [-2, 2]. Figure 6.10 shows the distributions of these scores before and after the camp by condition, and Figure 6.11 shows the scores each day.

At the beginning of the camp, most of the campers' self-efficacy scores were low: the control mean was -0.54 and the experimental mean was -0.74 where 0 is neutral self-efficacy. These distributions of pre-camp self-efficacy scores were not significantly different (H=0.87, p=0.351). After the camp the combined programming self-efficacy scores were higher for both groups, but the

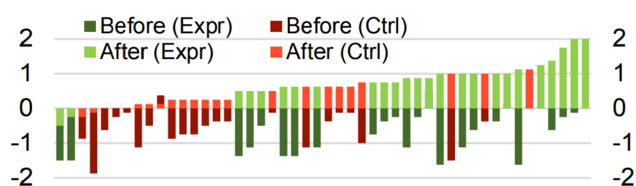


Figure 6.10: Aggregate self-efficacy scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Self-efficacy increased for most campers, but increased significantly more in the experimental group.

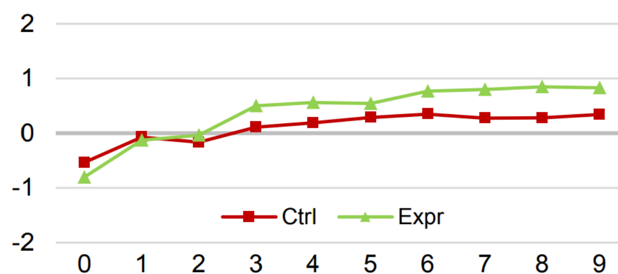


Figure 6.11: Mean self-efficacy in each group for each day of the camp. The experimental campers' self-efficacy increased after the introduction of the intervention (day 2) and ended mildly positive, while the control campers' ended at a neutral level.

experimental group's self-efficacy was significantly higher than the control's (H=12.2, p=0.0005), with a control mean score of 0.29 and an experimental mean score of 0.88. With a mean difference effect size of 0.59, the control group ended the camp with a neutral belief in their ability to create web applications, whereas the experimental group was unambiguously positive (shown in Figure 6.11).

When we considered the change in self-efficacy – computed as the difference between the last and first days' combined scores – the differences were even more substantial. The control group's change in self-efficacy score was a mean of 0.90, whereas the experimental group's change in self-efficacy score was a mean of 1.61, leading to significant effect size of 0.71 increase in self-efficacy (H=14.1, p=0.0002). These results show that the problem solving intervention in the experimental group likely had a strong positive effect on campers' beliefs in their abilities to successfully code interactive web sites.

Another notable difference was the self-efficacy changes by gender: after the camp, many male campers still had negative programming self-efficacy, as did many female campers in the control group, but all female campers in the experimental group reported positive self-efficacy.

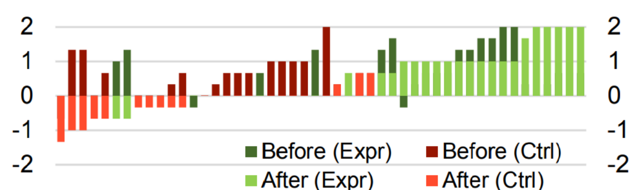


Figure 6.12: Aggregate growth mindset scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Growth mindset stayed positive in the experimental group but turned more negative in the control.

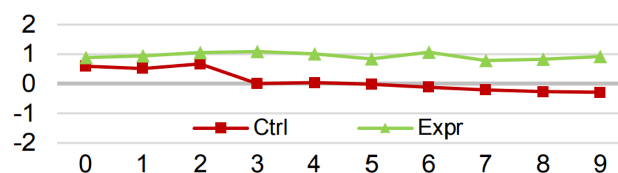


Figure 6.13: Mean growth mindset score across the ten days, by condition. While the control campers' growth mindsets deteriorated sharply when JavaScript was introduced (day 3) and continued to degrade, the experimental campers maintained their existing growth mindset throughout the camp and showed a slight upward trend.

6.4.6 *Impact on Growth Mindset*

As shown in recent prior work, introductory computer science courses can erode growth mindsets, making students believe that general aptitude is inborn and cannot change [38]. We hypothesized that by increasing campers' success at problem solving and programming, we could prevent this erosion of growth mindset. Similar to the self-efficacy measure, we tested this hypothesis by mapping campers' pre-camp and end-of-day growth mindset survey responses to a [-2, 2] scale, then took the mean of the responses. Because the survey measured fixed mindsets, we negated the value, so that positive values indicated growth mindset and negative indicated fixed mindset.

As shown in Figure 6.12, at the beginning of the camp, the campers in both groups began with a comparable but weak growth mindset, with the control group having a mean of 0.60 on our scale (slightly below "agree" on our scale), and the experimental group having a mean of 0.87 (also slightly below "agree"), (H=1.89, p=0.169). After the camp, however, the campers in the groups were significantly different (H=21.9, p=0.000003): the control group campers' mean score was -0.20 (meaning their growth mindset had eroded to a slight fixed mindset, replicating prior work [38]), and the experimental group's mean score was 0.93 (a moderate growth mindset). Figure 13 illustrates this trend over time, showing that the campers in the experimental group maintained

their belief that aptitude can improve while the control group campers' growth mindset eroded. The change in growth mindset scores for each camper from before and after the camp (the difference between last and first day's combined growth mindset score) was also significant (H=6.20, p=0.012). The control group had a mean decrease of 0.80 in their growth mindset, whereas the experimental group only had a 0.07 mean decrease.

6.5 Discussion

Our results provide some of the first evidence that teaching problem solving for programming is not only possible, but can improve productivity, promote independence, increase self-efficacy gains, and reinforce growth mindset in a learning setting where it typically erodes greatly. Moreover, the trends we observed are consistent with the intended mechanisms of our intervention: campers in the experimental group were significantly more likely to recall and describe the strategies they employed and more likely to request help with a problem after they had already attempted to solve it. Although our experiment did not allow us to separate the relative contributions of our four interventions, our results suggest that they worked together to teach and reinforce the idea that awareness of one's strategies and their effectiveness is critical to successful programming.

In one sense, these results are what general theories of problem solving, self-regulation, and metacognition would predict. Prior investigations into metacognition instruction generally provide students with a domain-specific problem solving knowledge and goal structure, plus incentives to learn from and avoid common metacognitive errors, such as poor or failing strategies [95] – and this is what our intervention did. The increased independence with which the campers in the experimental group worked would also explain why their self-efficacy increased, and possibly why their productivity increased: if they were more effective at recognizing effective and ineffective strategies through increased awareness, they would have made more progress on problems without having to wait for help from the camp helpers. In contrast, the campers in the control group,

like students in most introductory settings, were usually stuck at the beginning of problems and required help to proceed. This may have reinforced that they did not "get" programming, weakening self-efficacy and eroding growth mindset.

If our interpretations are correct, our study has important, far-reaching implications for how we teach people to code. First, given the strong positive impact of growth mindset on lifelong learning [12] and the tendency of introductory programming settings to weaken it [38, 102], if our findings are replicated and further substantiated it would arguably be unethical for learning technologies and teachers to not adopt some form of instruction on problem solving. Designers of introductory programming learning technologies such as Scratch, codecademy.org, and code.org could embed explicit instruction about the problem solving stages we propose, perhaps even finding ways to detect what stage a learner is in and offer constructive feedback about strategies and tactics to proceed. By incorporating such instruction, we might also increase participation in computing by women and ethnic minorities, who often start with lower self-efficacy or fixed mindsets in computing settings [35].

There are still several open research questions about our intervention. Future work should explore which aspects of the intervention were most responsible for the effects. There are also wide-open design questions about how to adapt the spirit of our intervention to other settings, including online learning technologies, and classrooms of various sizes and structures.

Part of this future work is also overcoming the limitations of our initial investigation. Studies should explore the effects of similar interventions on other age groups, levels of academic achievement, and other socioeconomic statuses. In particular, the campers we recruited mostly came from high socioeconomic status families in a mostly white city, and had a high likelihood of knowing someone who worked in the software industry in some capacity. Viewing technology from an amplification lens [110], our results could have been quite different in rural or low socioeconomic settings, where prior work has shown self-efficacy and exposure to computing to be substantially

lower. Future work should also replicate the effects of our study with other instructors, other programming languages, other problems, and other cultures. For example, we achieved these effects with a team of energetic but novice teachers; achieving them with more experienced teachers may require different approaches. Our work also did not explore the extent to which the changes in self-efficacy and reinforcement of growth mindset are robust to time: it may be that the campers' self-attitudes were shaped contextually and not generalizable to other settings.

In addition to generalizability concerns, the time of day difference in our camps may have caused internal validity issues. Because the experimental group was in the morning, it might have attracted higher achieving campers not deterred by a 9 am start time and may have received higher energy instruction from teachers and helpers, unlike the afternoon, which occurred after lunch and a long morning of instruction. We tried to overcome this confound by placing the experimental group first, ensuring that the control group also received many benefits from the second delivery of the camp (fewer technical problems, improved answers to requests for help, clearer delivery of direct instruction), but it is possible these advantages did not outweigh possible bias. Limitations aside, if we can repeat these findings broadly and deepen our understanding of how to teach problem solving in programming in a wide range of contexts, there is broad potential for impact on the world's current efforts to teach programming. Dozens of countries have begun initiatives to teach programming in K-12, and hundreds of companies have started coding boot camps. Our results suggest that problem solving instruction can and should be an instrumental part of them. If we can train the teachers, develop the materials, and adapt the learning technologies to empower learners to understand and solve programming problems, we might just meet the ever-growing demand for a diverse and computationally literate global society.

Chapter 7
THE PROBLEM SOLVING TUTOR

Chapter 6 investigated one possible approach to explicitly teach and scaffold the development of metacognitive awareness in a classroom setting. In this chapter I present the *Problem Solving Tutor* (PSTutor), a novel interactive computer-based tutor for teaching programming self-regulation skills that 1) works at arbitrary scale, 2) only requires a teacher to author reusable content, and 3) and can be used to to support instruction on all possible programming activities and scenarios. In the following sections I discuss the design of the PSTutor, demonstrate how the PSTutor supports the development of metacognition and self-regulation skills, and describe how content is created for the tool. I then present an empirical evaluation of the PSTutor in a web development course where learners reported PSTutor having several positive impacts on their process and self-regulation. Additionally, I'll show that students who viewed PSTutor scripts for a programming assignment were significantly more likely to earn higher grades on the assignments. This chapter contributes two things; 1) a new genre of programming tutorial that reveals the self-regulation involved in effective programming problem solving, 2) evidence of this genre's effect on successful programming.

7.1 Problem Solving Tutor

Rather than attempt to explicitly teach metacognition and self-regulation, the PSTutor is designed to *model* high quality self-regulated programming skills. The prior work on modeling self-regulation discussed in Section 2.4 shows that while many existing media have the capacity to model self-regulation in programming, they impose several limitations on authoring and learning:

- Worked examples and textbooks have limited capacity to show programming process and

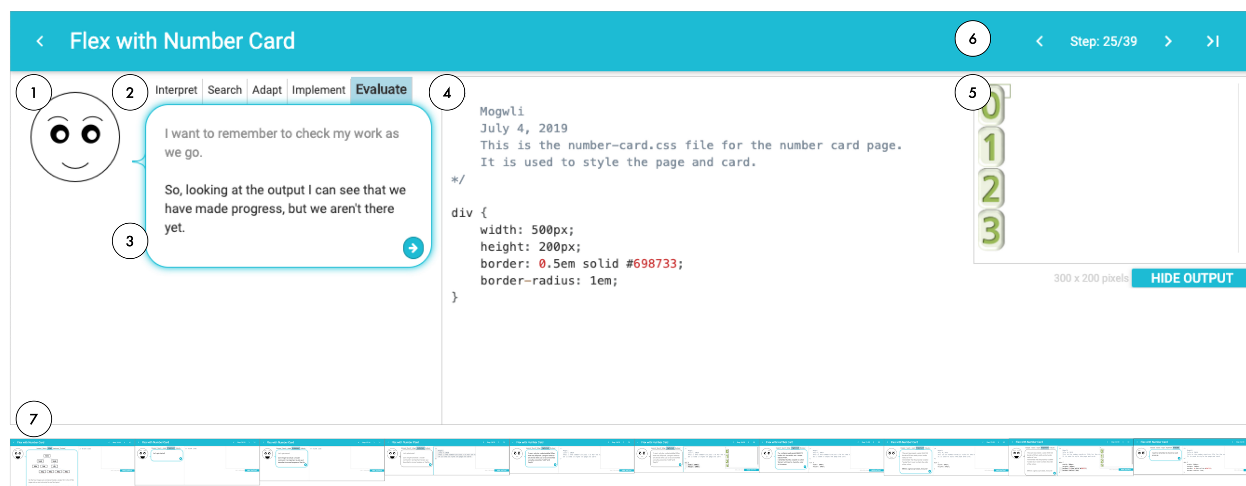


Figure 7.1: The Problem Solving Tutor interface, showing 1) the programmer, 2) the programming stages, 3) the speech bubble, 4) the code area, 5) the program output area, 6) the navigation controls and current step, and 7) 10 steps from this script's 39 steps. This is script was written for assignment A2a in our evaluation.

self-regulation over time, and to show the up-to-date program output.

- Programming demonstrations can be challenging to perform, record, and edit.
- Tutors and tutorials primarily offer opportunities to practice programming, not to see other programmers modeling effective self-regulation.

To address these limitations, I invented a new medium, the Problem Solving Tutor, which represents both a *scripted modeling of programming process* and the *web-based viewer* of these scripts. Figure 7.1 shows a PSTutor script presented within the viewer. PSTutor scripts, which are composed of a series of steps (as shown in the sequence in Figure 7.1.7), are designed to explicitly model the self-regulation of programmers, allowing learners to observe and step through carefully crafted scenarios depicting a programmer thinking aloud while they solve a programming problem. Unlike existing media, they are temporal, show program output, are easy to edit and navigate, and explicitly support modeling of metacognitive and self-regulatory behaviors.

Property	Data type
Type	`Code` | `Speech` | `Assess`
Content	*String* (code, speech, or assessment prompt)
Emotion	`Neutral` | `Annoyed` | `Shucks` | `Happy` | `Excited` | `Thinking` | `Surprise` | `Wonder`
Stage	`Interpret` | `Search` | `Adapt` | `Implement` | `Evaluate`

Table 7.1: The metadata required for each step in a PSTutor script.

Figure 7.1 illustrates our approach, showing the PSTutor. The figure portrays a snapshot of a programmer evaluating their progress on problem solving. The presented tutorial also models mistakes, recovering from them, and identifying making and reflecting on decisions. By modeling programmers' self-regulation in this way, our goal is to show a novice programmer how other programmers think while they code, in the hopes that they transfer that thinking to their own future programming.

This approach to teaching self-regulation builds upon social cognitive theory [7]. The central concepts in social cognitive theory are that people acquire knowledge by observing others within a context of social interactions, experiences, and environments and that by observing other people modeling a behavior, people build the metacognitive knowledge structures that guide their later practice. Prior work shows that self-regulation skills emerge from social origins [101], particularly observations of others' behavior combined with social guidance. PSTutor provides exactly these things, helping learners build the knowledge structures for action in programming by modeling programmers' reflections, decisions, and actions.

7.1.1 The Script Data Structure

The essence of the PSTutor medium is the data structure schema that defines its content. Overall, the data structure for a single script is quite simple. It contains a name, a description of the programming

scenario it portrays, the initial state of a program source file, and a list of *steps* that convey the sequence of a programmers' thoughts and actions while editing that source file. Each step consists of the metadata listed in Table 7.1, including the type of step, textual content for the step (speech, code, or assessment), the emotion associated with the step, and the programming stage in which the step occurs. In addition to the content in Table 7.1, `Code` steps must also specify a text buffer index to determine where code is inserted or deleted, and `Assess` steps must specify a list of multiple choice options, and for each, whether it is correct and explanations for why or why not. The data structure does not provide explicit support for program execution. As we describe later, that is a feature of the PSTutor viewer, which is responsible for executing programs in a browser and displaying their output.

7.1.2 Five Kinds of Self-Regulation Content

There are five major kinds of content that PSTutor scripts can contain, each conveying a different aspect of self-regulation.

Providing process context

One important thing to teach is the contexts in which various programming behaviors happen. Following from the problem solving framework detailed in Section 3.1, the scripts identify when each programming behavior is occurring in the script. These behaviors include *Interpreting* the prompt, *Searching* for analogous solutions, *Adapting* previous solutions, *Implementing* a solution, and *Evaluating* an implementation. To teach these programming contexts PSTutor scripts require explicit markers indicating when the programmer has switched behaviors. In the PSTutor we identify the times in which these programming behaviors happen as "stages".

Understanding these stages is critical for situating which stage certain kinds of thinking and work happen in. Consider a programmer exclaiming, "*Ugh, I don't understand this at all!*" Without

Step	Stage	Speech Content
13	Interpret	It is important to keep in mind that there are two sides of the assignment (client and server) that are very much interconnected.
14	Search	Let's think about similar problems we have encountered. We have experience with client-side DOM manipulation, event listening, and fetching from web services which will all come into play for the client side development. Let's jot down a few examples we may want to reference during this project.
15		*Makes code comments identifying where to find examples and what technologies they are using.*
16		In class we learned how to listen to a port, define routes, interact with the response and request objects using Node JS and the framework Express. Let's note the example slides and other resources we may want to reference here too.
17		*Makes code comments identifying which lecture slides to consult.*
18		Now that we have identified the problem and some resources to consult, let's consider how we want to approach the problem, building off our past experiences.
19	Interpret	When doing client- and server-side development there are several approaches that I can think of taking. Let's make a (non exhaustive) list so we can think about which we would prefer.
20		In no particular order, here are the four common approaches I'm thinking of: 1. Complete the client side first, then the server side. 2. Complete the server side first, then the client side 3. Lead with the client side, but build the server side along the way 4. Lead with the server side, but build the client side along the way
21		Remember, with any of these approaches, we should be iteratively developing our code, checking and debugging as we go.
22		Let's explore what we think the pros and cons of each approach are in regards to this Pokedex project!

Table 7.2: Steps 13-22 of a script written to support the A4 assignment of the evaluation, showing the speech bubble text. Italicized steps are code edits.

context of what stage they are in and what content they are paying attention to, it is difficult to know what programming behavior the programmer is engaging in. These kinds of decontextualized statements often occur in livestreamed programming sessions, for example, where a developer might make such a remark offhand, without sufficient context for a viewer to understand what they are trying to understand. In contrast, providing explicit markers about what stage the programmer is in, as PSTutor scripts does, clarifies the intent of the remark. When a programmer is *Interpreting* a problem such a remark would mean that they do not understand what they are being asked to do. Or, if they were in the *Evaluating* stage it would be clear they understood the problem enough to implement a solution and are struggling to understand the cause of unexpected output.

To visualize the stage the programmer is in, the PSTutor interface portrays these stages as a

persistent list of stage names, as shown in Figure 7.1.1. The PSTutor highlights the current stage and animates the highlight moving to a new stage when the script calls for a transition between stages to draw learners' attention to this change.

Modeling thought

Programmers' thoughts, unless somehow externalized, are invisible to an observer: code examples cannot show it and recorded videos of programming only have small glimpses via think-aloud. Yet, based on social cognitive theory [8], it is critical that a learner be able to observe the thinking process of programming in order to learn how to think in the same way. PSTutor scripts can model the thinking of the programmer by including explicit think-aloud statements in scripts, which are then displayed in a speech bubble (see Figure 7.1.2). The speech bubble simulates a think-aloud protocol, exposing the programmer's internal dialogue to the learner, but without the burden of an actual person having to think-aloud while they program.

PSTutor scripts can express many kinds of internal thought using the speech bubble. Thoughts expressed could include statements demonstrating metacognitive awareness of their process (e.g. Table 7.2.14), or explicitly express rationale for decisions (e.g. Table 7.2.19). The speech bubble is also important for explicitly modeling self-regulation skills like monitoring process (e.g. Table 7.2.18) and strategy selection (e.g. Table 7.2.22). In addition to these, and other, types of internal thoughts, the speech bubble can express other important context like expressions of emotion ("*It worked! I'm so happy!*") or any other content that can be expressed or communicated through text.

Modeling emotional state

Programming is not just about process and thought; emotions also play an important role and influence motivation and decisions [5]. Understanding what emotions programmers might encounter, and how to manage them, might allow learners to set proper expectations for their future programming.

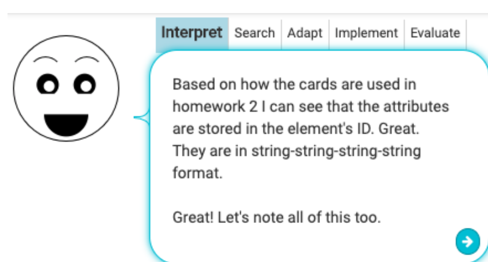


Figure 7.2: A step showing the *excited* emotion.

This could help to increase their programming self-efficacy and help them adopt an identity that includes programming. By demonstrating the roles of frustration, confusion, and other feelings in authentic programming contexts, learners get to experience how other programmers handle these situations. However, like thoughts, the emotional state of a programmer at work may be invisible to an observer. For instance, encountering a bug can cause frustration and adversely affect motivation while coming up with a novel idea for a solution can be exciting and increase motivation. Yet, few prior media explicitly convey these emotions beyond the excitement or frustration in scripted instructional videos.

PSTutor scripts can model emotions either through the speech bubble and through facial expressions. To model emotions with facial expressions, PSTutor uses an avatar to represent the programmer. The avatar can express emotions through a range of facial expressions, listed in Table 7.1. Figure Figure 7.1.3 and 7.2 show the *excited* and *happy* expressions in context, respectively. We included the negative emotions *annoyed* and *shucks* to help set expectations and show that programming may, at times, be frustrating and spark negative emotions like the confusion and frustration that novices often face [13]. By setting the proper expectations of programming the PSTutor may be able to support a growth mindset and support greater self-efficacy [13]. Demonstrating the negative emotions to the learner also allows for showing that negative emotions can be acceptable and can be overcome.

Modeling programming

Modeling the thoughts and emotions of a programmer is important, but lacks meaning without connecting them to how the programmer actually writes and edits code. Showing the generation of code as an output of the thinking allows a learner to understand how, and why, a programmer is writing or modifying a specific part of a program. PSTutor scripts convey changes to code by explicitly representing character-level edits to a single file program, allowing scripts to have a fine-grained representation of the transformation of a blank file into a finished program.

The PSTutor shows code modifications through a read-only editor and output pane (see Figure 7.1.4). To attract the focus of the user, the editor animates character-level "typing" of code by the programmer during implementation steps; we included this typing animation to prevent recently discovered stereotype threats about typing speed [39] from harming learners' programming self-efficacy; it also allows script authors to easily encode things like typos and backspacing repairs to signal that no programmer types perfectly. The PSTutor also highlights the most recent block of code written for easy identification, even after proceeding to subsequent script steps.

Additionally, the output pane (Figure 7.1.5) can show the output of the current program in the editor. This combination of features allow for many kinds of illustrations of programming actions like code writing, editing, deleting, backtracking, or any other code manipulation. The live updates in the program output pane make it easy to illustrate rapid cycles of testing and editing as well. The current iteration of the PSTutor includes a JavaScript parser which will automatically display live output in the output pane. Future development will support other languages. However, when authoring the script, any type of visual output, from any programming language, can be supported by providing images of the output to be displayed in the output pane.

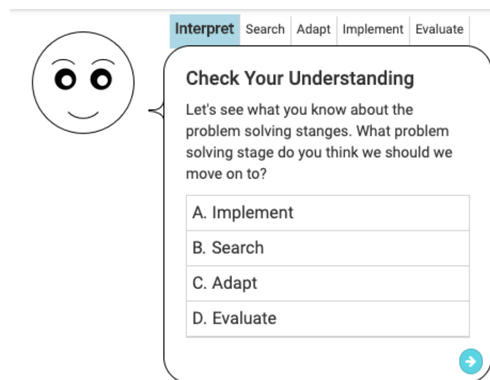


Figure 7.3: An assessment step in a PSTutor script, prompting the learner to reflect on what stage the programmer should move to next.

Modeling Self-assessment

To ensure learners are attending to the script's content, the PSTutor can present formative assessments at any point in a script. Formative assessments have been known to be beneficial to learning [97] and have been successfully integrated in to other learning technologies for programming [58]. While all of the content in a PSTutor script is intended to explicitly model a programmers process and self-regulation, explicitly assessing learners knowledge of process and self-regulation can help reinforce these skills in context. Assessments can punctuate important concepts in the script (e.g. problem decomposition), bringing the learners' attention back to the stages (as in Figure 7.3), or to break up scripts with many steps.

7.1.3 Script Authoring

The PSTutor is only as useful as the scripts that authors write. Therefore, I provide two types of support for authors: an authoring tool to remove barriers to constructing scripts and authoring guidelines to ensure their quality.

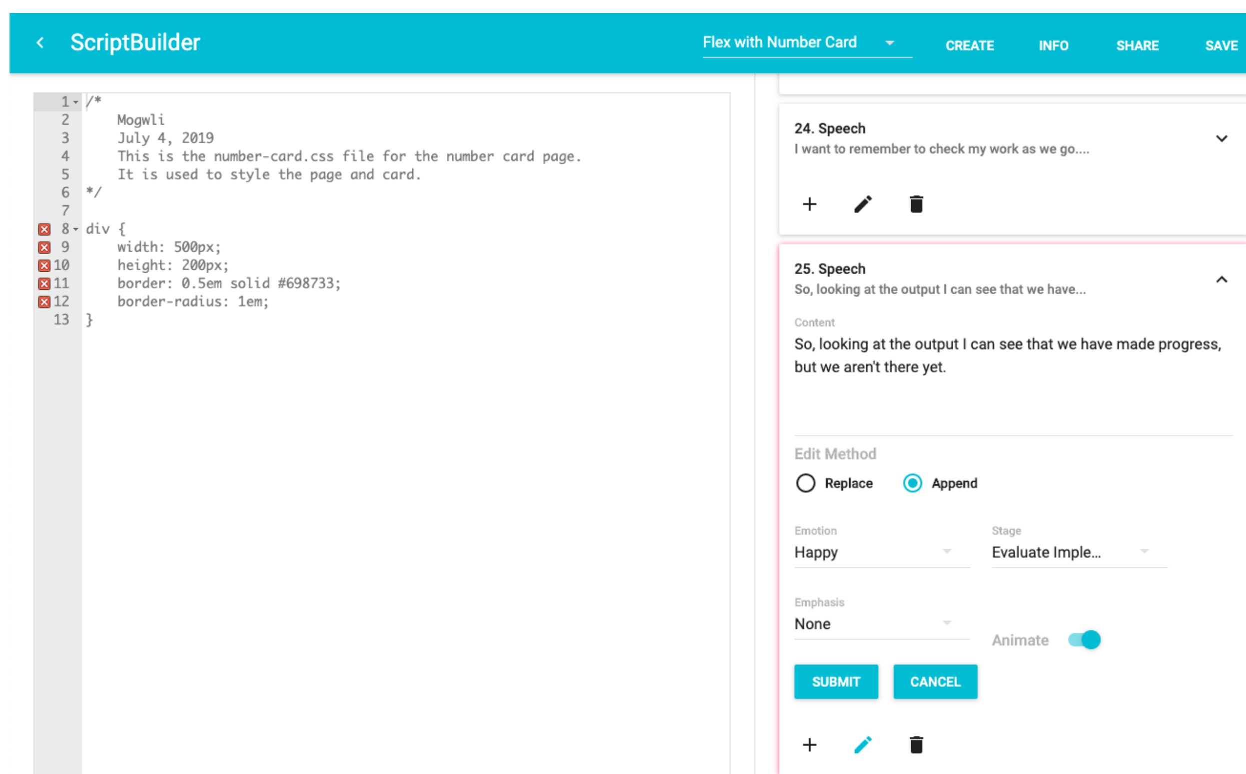


Figure 7.4: The PSTutor authoring interface, showing the current state of code on the left and the sequence of script steps on the right.

Authoring Tool

Authoring a script by writing a list of steps with the metadata in Table 7.1 is generally straightforward. Most of the content to write is text; the hard work is envisioning an instructive scenario and considering the frequency and depth to attend to internal thoughts. However, there are a few challenges in script authoring that need additional support. First, code in a script step is represented by a series of inserts and deletions at specific source file indices; while this format makes it easy for learners to see the actions a programmer performs in the script, it makes it very hard for script *authors* to see the state of a program at any given point in a script and know exactly what code is being inserted or deleted. Second, while authoring scripts in a basic markup format (e.g., a JSON

Guideline	Rationale
Show all steps.	Early stages of social learning require highly granular modeling of individual steps [101].
Use a sociable and supportive tone.	Such tone helps frame failure and uncertainty as necessary and potentially productive [77].
Include the learner (e.g use "we" pronouns).	Learners must feel involved in the modeling to ultimately form self-efficacy [77, 101].
Identify plans.	Prior work shows that modeling programming as an activity that requires planning improves outcomes [76].
Explicitly identify what is being modeled.	Successful modeling requires attending to the behavior to be modeled [70].
Identify transitions between stages.	Prior work shows that recognition of programming stages promotes more structured programming process.
Identify the problem being solved.	Prior work shows novices struggle to remember to carefully interpret programming problems before coding [115].
Identify ambiguities.	Learners can interpret confusion as failure [101]; this prevents self-efficacy erosion.
Reflect on decisions.	Authentically models and frames programming as requiring careful analysis of problems and decisions [59].
Model evaluation of decisions.	Models programming as an activity requiring reflection and reconsideration of decisions [59].

Table 7.3: Evidence-based guidelines for authoring PSTutor scripts.

data structure or an XML format) is feasible, such formats make it hard for authors to imagine what a learner would see at each step.

To address these two problems, we designed the authoring tool shown in Figure 7.4. It makes it easy to add, modify, and remove steps, to see the state of the program being constructed at any point in a script, and easy to quickly see a preview of what a learner would see at any stage.

Authoring Guidelines

While the authoring tool makes it easier to construct PSTutor scripts, writing content is still the primary challenge of authoring. In the course of writing many PSTutor scripts for many learning contexts, we have arrived at several theoretically-informed authoring guidelines. We list these in

Table 7.3. The general principle of these guidelines is to make everything explicit, calling attention to precisely the aspects of process that the modeling intends to portray.

Table 7.2 provides an example of how these guidelines can be met. For instance, the dialogue in this table *shows all steps* of the programmer's thinking while in the *search* stage. This script could go into more detail about what examples to select for reference and why to select those examples. The nuance around the level of detail to model for each behavior is the hardest part of deciding what content to include while authoring the scripts. Table 7.2 also shows the use of *sociable and supportive tone* expressed through the exclamation in Table 7.2.22 and the entire dialogue *includes the learner* using "we" when considering experiences and actions. An example of *identify plans* can be seen in Table 7.2.19 and Table 7.2.22. Similarly, Table 7.2.14 *explicitly shows what is being modeled* (the searching stage) and, like Table 7.2.18, also *identifies the transitions between stages*. Table 7.2.19 *identifies the problem* that the programmer is addressing in this script (this is done more explicitly in earlier steps).

Following these guidelines, we have authored many types of PSTutor scripts, including scripts showing a programmer struggling to understand HTML tags, creating styling for HTML elements using CSS, using the HTML5 canvas API to create static and animated drawings, and creating animations that respond to keybindings.

7.2 Evaluation

The central design hypothesis of PSTutor is that *by modeling programming self-regulation, learners will more productively self-regulate their own programming, and therefore be more successful at programming*. To test this we conducted a classroom study in which we provided students in a programming class relevant PSTutor scripts before programming assignments. This allowed us to investigate authentic student interactions with the self-regulation content in the environment in which they are expected to develop these skills. To understand potential impacts on self-

regulation behaviors, we conducted a survey after each programming assignment asking about their programming process. To understand potential impact on programming outcomes, we gathered the grades on each assignment. Below we detail the course, the assignments, and the scripts we presented, then present our results.

7.2.1 Learning Context

The course we chose was a front-end web development course that taught HTML, CSS, JavaScript, and Node.js. The course was typically taken by freshman and sophomores after an introductory programming course. We partnered with the instructor of one section of the course taught in summer 2019. The course included 37 undergraduate students (13 identified as women, the rest as men); all had passed at least one introductory programming course covering variables, loops, functions, and arrays. The course included six assignments of increasing difficulty:

- *Assignment 1*. Create a web page using HTML/CSS, adhering to visual references.
- *Assignment 2a* (A2a). Use HTML/CSS to create the visual elements and user interface for a card game called "Set!"
- *Assignment 2b* (A2b). Use JavaScript to implement the game rules for the "Set!" game.
- *Assignment 3*. Use AJAX to fetch text and JSON formatted data from an API and display the results in a "Pokedex" format.
- *Assignment 4* (A4). Use Node.js and Express to implement a web service, then building a client to retrieve the data from the web service and display it.
- Final project. Develop the front and back-end for an e-commerce store of the student's choosing (real or imagined).

At the university at which this course was offered, students in summer courses often take only one or two classes while working part time jobs. Therefore, students likely had similar lives outside of class as during the normal academic year, just with fewer classes.

7.2.2 PSTutor Scripts

For our intervention, we authored PSTutor scripts that reflected the work students would be asked to do in A2a, A2b, and A4:

- A2a. This script depicted a programmer styling existing HTML elements to meet a visual goal. The final output is a refined version of the output depicted in the output view of Figure 7.1.

- A2b. In this script the programmer demonstrated the development of the "isASet()" JavaScript function that the students received as starter code for the assignment. This function checked to see if a set of selected HTML elements all have the same attributes (stored as strings in the ID of the element), or that none of them have the same attributes.

- A4. This script modeled a programmer preparing to develop a full client-server application. In this, the programmer models identifying server specifications based on known client needs, identifying resources to consult for reference, and considering the trade-offs of developing the server or client side first, or developing them in parallel. This script contained no code, but used code comments to capture plans, resources, and pros and cons for different approaches to development. Table 7.2 shows the dialogue for steps 13-22 of this script.

For each of these scripts, we carefully followed all of the PSTutor authoring guidelines presented earlier in Table 7.3.

We provided the scripts at the same time that students received the assignment details; the assignments instructed students to refer to the tutorial before completing the assignment.

7.2.3 Data Collection

Instructors graded the assignments based the students' submissions passing test cases. Instructors also assessed the quality of the code, deducting points if students' submission failed to follow a specified code style, had redundant code, misused the HTML DOM, used improper naming conventions, or lacked adequate code documentation. After the course concluded, the instructors provided the authors a list of anonymized participant IDs and their associated grades.

As part of each assignment submission, we asked students to reflect on their PSTutor use when turning in A2 (A2a and A2b were part of the same assignment) and A4. We asked students to answer the question, "*While you were programming, how did you consider, or use, the problem solving stages or examples from the Problem Solving Tutor? How did it work for you?*"

The instructor of the course incentivized the use of the PSTutor and the reflections by offering 2 bonus points. We also logged when students used the PSTutor collecting the time and direction of each step (moving forward or backwards) through each of the scripts.

Theme	**Description**
Better structure	Student reported being, or being reminded to be, more systematic because of the PSTutor.
Better awareness	Student reported being more aware of their process.
Better decomposition	Student reported PSTutor reminding them systematically decompose problems.
Better planning	Student reported PSTutor helped them plan their solutions.
Better caution	Student reported the PSTutor reminded them to slow down to prevent errors.
Process alignment	Student reported that PSTutor's stages aligned with their current process.
Process friction	Student reported resistance to following the structured processes modeled by the PSTutor.

Table 7.4: Themes in students' self-reported impact of the PSTutor scripts on their process and self-regulation.

7.2.4 Results

To begin, we consider evidence of PSTutors impact on students' self-regulation behaviors. Overall, 18 of the 37 students in the class submitted survey responses with their assignment submissions, for a total of 35 submissions across the 3 assignments (one participant reflected on assignment 2 but not on assignment 4). To analyze the survey reflections, two authors independently and inductively identified themes around how the participants reported the use of the PSTutor affecting their programming process. The authors compared themes and reconciled differences to create a final codebook. Each author then independently coded each response, disagreeing on 5 responses. The disagreements were primarily due to the difficulty in distinguishing between the Awareness and Structure codes. These were discussed and reconciled, reaching 100% agreement. (We adhere to Hammer and Berland's perspective on qualitative coding [43], which treats the results of our coding effort as organizations of claims about data rather than data in and of themselves.)

Table 7.4 shows the themes that emerged from our analysis. Overall, beliefs about the value of PSTutor to their process were primarily positive, and focused on the impact the behaviors modeled in the scripts had on their own programming processes. For example, one student's reflection noted a *better structure* in their process, *better awareness* about their process, and *better planning* of their work:

> "I was inspired by the sources that is given to me. I became more aware of what I was doing instead of blindly follow the spec and write the code. I started to plan a little bit before I write any code and I use comments and a function skeleton as a map for my code."

Another student appreciated the detailed modeling of planning, which they believed led to *better planning*:

> “I think how it led me to planning my program is great. If I read the spec really carefully once and take notes and make plan, I think I would run into a lot less problems when I test my code.”

Other students mentioned the impact of the PSTutor scripts on *better decomposition* of their problem, relying on the tutor in the middle of their process to self-correct poor planning:

> “I didn’t think I would need to use the PSTutor, and I started working on the homework without considering it. Then eventually when I got stuck, I checked back and went through the steps to see how it could help me. It reminded me how to structure and piece out my code. It helped me rethink the kind of functions I needed to accomplish what tasks.”

Some students mentioned that the PSTutor scripts *aligned* well with their existing process, but helped them better practice *caution* in their work:

> “...the problem solving stages are really practical, because that is basically how I got my homework done... It turns out I would consider this as a repetitive steps where I go from implement, to search, to adapt, test, and start all over again for each function. These help me to reduce errors while programming.”

Some students resisted the tutor’s modeling of self-regulation indicating *process friction*, instead using highly iterative approaches:

> "The problem solving stages from the problem solving tutor does not apply that much to my situation. When I am coding, I usually code as I read the specification instead of stop, write some comments for planning, then move on to coding. Since I check my program frequently enough to spot some errors ahead of time, it is faster for me to just go ahead and code right away instead of spending too much time on reading. I believe the best way to figure out that I actually need help with something is by implementing the actual product, because it is a different story than just reading and understanding the text."

While the self-reported impact on self-regulation suggested largely positive impacts, this did not necessarily correspond to actual changes in students' self-regulation behaviors, and even if it did, it did not necessarily translate into success on programming assignments. To check for this impact, we analyzed PSTutor usage logs against assignment grades. In total we collected logs from 22 students who used the PSTutor. We counted the *number of steps* viewed by each student.

	With PSTutor	**W/out PSTutor**	**U**	**p**	***d***
A2a	[7,9,10] / 10	[3.5,8,10] / 10	75.5	0.01	0.47
A2b	[10.5,19,20] / 20	[0,14.5,20] / 20	63.5	0.003	0.56
A4	[8,19,24] / 25	[0,5,23] / 25	73	0.008	0.49

Table 7.5: For each of the three assignments, the [minimum, median, maximum] scores out of the maximum possible score, split by students who did and did not use the PSTutor, plus Mann-Whitney U statistics, p-values, and Cliff's delta effect sizes comparing the groups.

Exploratory data analysis of the association between using PSTutor usage and student grades suggested a few trends: 1) students either viewed entire PSTutor scripts for all three assignments or did not view any of them, and 2) there appeared to be an association between viewing the scripts

and higher assignment grades. Therefore, we decided to compare two groups of students: those who had viewed at least one complete script and those who had not viewed any; this distribution was bi-modal, with only one student failing to complete viewing a script (skipping the last half of one script). Table 7.5 shows the distributions of each group for each assignment, and the Mann-Whitney U, p-value, and Cliff's delta [27] effect sizes for each comparison. On each assignment, students who use the PSTutor scored significantly higher on the assignment than students who did not. Effect sizes suggest a moderately high probability that a student viewing PSTutor content would do better than a student that had not.

7.2.5 Limitations

Our results have several limitations. First and foremost, because we chose a field deployment and not a controlled experiment, our evidence of causality is limited to students' self-reported changes in their programming process and self-regulation behaviors. There are therefore a number of possible confounds. The students who chose to view the PSTutor scripts might have been the students with strong self-regulation skills, and those might have caused the stronger performance on programming assignments. It is also possible that grades on the assignment were largely explained by other factors such as prior programming knowledge or procrastination, and that these factors were correlated with stronger self-regulation skills. Because we partnered with a busy instructor, we had limited access to measurements of these confounding factors.

While our qualitative data strongly supports an interpretation that the PSTutor *did* cause behavior change, this is weakened by the fact that not all students submitted the post-assignment reflection surveys. It is possible that had all students submitted the survey, many of the students who performed poorly on the assignments would have reported similar positive sentiments about the PSTutor script's impact. This would suggest that either other factors were responsible for assignment grades, or that the PSTutor did have an impact, but it was a smaller factor in grades.

7.3 Discussion

PSTutor establishes a new genre of scripted modeling of programming process capable of scaling the instruction of self-regulation in programming. Our evaluation showed that students believed the PSTutor scripts they viewed helped them think about their programming process in many new ways, from simply being more aware of their process and reminding them to slow down, to being more careful with their coding and prompting planning and problem decomposition. Our analysis of the PSTutor usage logs and student grades showed that students who used the PSTutor achieved better grades on the corresponding assignments than those who did not.

Together, these results suggest that scripted modeling of self-regulation in programming, in the context of a programming course, can influence student thinking and self-regulation behaviors. These are potentially transformative results considering prior work in computing education has not demonstrated any meaningful impact on self-regulation at scale, and self-regulation is such a large factor in programming success. However, it is important to reiterate that our evidence of causality is limited. It may be that the PSTutor improved self-regulation and increased assignment scores, but there are other interpretations. For instance, the most likely alternate explanation is that students with strong self-regulation skills tended to use PSTutor, and students without such skills tended not to. Similarly, only 18 of the students submitted reflections on their PSTutor usage; the 4 students who used the PSTutor but did not submit a reflection may have had very negative experiences that went unreported, skewing our perception PSTutor interactions.

While our causal evidence is limited, our results are consistent with predictions of social learning theory: by modeling important self-regulation and programming behaviors students were able to observe and begin to emulate those behaviors. Our study provides some insights into possible causal mechanisms. Based on the student self-reports listed in Table 7.4, it could be that some students simply were not aware of their internal processes and gained *better awareness* of their processes by using the PSTutor; in essence, they were starting from no self-regulation skills at

all, and the PSTutor scripts helped them acquire some basic skills, to great effect. Alternatively, students may have already been aware of their thought processes but by providing the language to *describe* various problem solving stages, the PSTutor scripts helped them recognize opportunities to be more systematic and cautious in their work. Another interpretation is that students had self-regulation skills already, and the PSTutor scripts just primed them to use them before working on their assignment.

These different interpretations have different implications for practice. For example, if PSTutor helps by priming, it might be that *any* prime to self-regulate would work, and that PSTutor scripts are more than is necessary. If PSTutor helps students see opportunities to improve their practices, it is not yet clear what role the content of a script plays in revealing these opportunities. If PSTutor helps students with no self-regulation skills start to develop them, how quickly do its benefits diminish? And if our results were confounded by self-selection bias, would a controlled experiment of the impact of PSTutor scripts still show positive results? Researchers should attempt to replicate and deepen our findings in different contexts, a greater diversity of users, and at a larger scale, to investigate the potential of PSTutor's approach.

Future work might also consider the potential impacts of improved self-regulation on self-efficacy, mindset, and identity. Stronger self-regulation skills might help learners develop an identity as a programmer, challenging the powerful gendered and racial stereotypes about who can be a programmer [66, 67], and lead to a greater diversity of people participating in computing.

The PSTutor presents an avatar that changes its expression to model being in a particular emotional state. However, little is known about how learners interpret or deal with emotion while programming. Future work should investigate how emotion interacts with the experience of programming, seek to understand the emotions that learners go through while programming, and develop methods to better set the emotional expectations of learners to bolster positive emotions and reduce or avoid negative emotions.

If the PSTutor does provide benefit to students, our results have many implications for educators. Novices are known to have poor self-regulation skills [44]; even our limited results suggest that incorporating PSTutor scripts into programming courses might impact student learning. Educators could allocate resources to writing scripts that align with their curriculum and educational practices. They might consider making self-regulation instruction mandatory, and assess it like other forms of programming knowledge. Educational technology designers might facilitate the development, collection, and sharing of scripted modeling materials making them easily accessible to a vast number of instructors and students.

Chapter 8

CONCLUSION

Computer science education is still a nascent field and, as such, it has only begun to employ knowledge of metacognition and self-regulation from other domains. While computing has long been connected to systematic thinking and cognitive processes, most research on the topic has focused on investigating the effect of training novices in specific metacognitive techniques and strategies or comparing the processes of novices and experts. The few that have attempted to use theories and techniques from domains such as learning science and psychology understandably apply them in the ways that have been validated in those domains. This has resulted in conflicting and surprising results [74, 75] and a dearth of theories that are informed by our knowledge of cognitive work from other domains but also consider the unique contexts that are specific to computing.

The goal of this dissertation is to advance computer science education by developing the domain specific nomenclature to discuss programming process and the mental work programmers engage in, establish a preliminary baseline understanding of novices' mental work from which to hypothesize, and invent promising pedagogical methods to support the development of metacognitive skills in novice programmers. It succeeded.

The frameworks presented in Chapter 3 provide a common language that allows researchers, practitioners, and learners to think about, reflect on, and communicate about programming process.

The studies of novices explored Novices' self-regulation in a laboratory setting and in authentic programming situations in Chapters 4 and 5, respectively. With this new understanding of where novices' mental skills might be lacking new opportunities for how to better support students became clear: Support the development of metacognitive awareness, followed by self-regulation skills.

The pedagogical approaches described in Chapter 6 and 7 offer two approaches to supporting the development of mental skills in novices. Their evaluations demonstrate that each approach has great potential for informing and enhancing the way computer science is taught.

Together, the previous chapters provide evidence to support the thesis statement:

> Novice programmers often lack disciplined programming problem solving, however, explicitly teaching and supporting the development of metacognitive and self-regulation skills can improve learners' problem solving, significantly increase productivity, self-efficacy, and independence, while avoiding growth mindset deterioration.

8.1 Future Work

The work presented in this dissertation is presented as a starting point for future work on the problem solving process for programming. There exists future work items common to all studies in this dissertation addressing concerns of generalizability including, but not limited to, replication of these studies in different contexts and with participants of other ages, levels of academic achievement, socioeconomic statuses, identities, and backgrounds. However, this section focuses on issues that reach beyond generalizability and offer future work to push computer science education into new frontiers.

8.1.1 Moving Beyond a Post-Positivist Paradigm

The work presented in this dissertation takes a primarily post-positivist perspective to provide a baseline set of expectations for future results. The significant results in this dissertation are presented in the aggregate, assuming some generalizability to novice programmers. However, there is also much to be learned from the individual experiences of the learners. Future work should adopt a more interpretivist approach and incorporate interpretivist methods to better elucidate the learners'

individual experiences, identities, contexts, values, and understanding of their programming process to begin building a robust picture of what learning to programming is like, and who is doing it.

8.1.2 Domain Specific Theoretical Frameworks

The problem solving and programming self-regulation frameworks presented in Chapter 3 are the foundations for a set of domain specific frameworks for programming, and expose a wide range of potential future work. Future work should build upon these frameworks both from a programming specific perspective, by identifying and incorporating more of the behaviors programmers engage in, and testing and adapting knowledge about learning and cognitive processes from other domains. As with any form of knowledge, future work should seek to improve the presented frameworks through validation and falsification, adapting existing frameworks or presenting novel ones to better represent our understanding. Future work should also seek to identify if, how well, and why cognitive theories from other domains apply to computing. It is only though this type of inquiry that domain specific cognitive theories will emerge and provide important new research areas in computer science education.

8.1.3 Understanding Novices' Self-regulation

The deeper our understanding of novices' mental work, the better future work can hypothesize and investigate how to best support learners in developing critical programming skills. Thus, future work should endeavor to deepen our understanding of the development of novice programmers' metacognition and self-regulation. While the studies presented in this dissertation leveraged many of the few techniques available to expose novices' mental work, future work should refine the training and instruments used in our studies to more accurately measure the mental work of novices. Future work should also seek to leverage the findings of awareness clusters and ordering of self-regulation skills detailed in Chapter 5 to investigate when, how, and why novices develop these skills.

8.1.4 Supporting the Development of Metacognitive Problem Solving Skills

The pedagogical approaches detailed in Chapters 6 and 7 are only two examples of how the development of mental skills might be supported. Future work on the Problem Solving Tutor should seek to adapt it to additional programming languages and contexts such as larger multi-file projects. Further development should explore opportunities such as gamification and representation such as allowing personalization of the avatar or allowing user agency through selection of which programming behaviors to explore next. Additionally, further development should investigate the difficulties in authoring content for the PSTutor and similar instructional content for modeling mental behaviors in educational tools and classroom settings. Researchers should use the results presented in this dissertation, and in that of prior work on metacognition in computing, to invent and evaluate additional methods to support students' mental work. These avenues of future work are likely to find that *how* problem solving is taught is just as important as teaching it at all.

It is my hope that the research presented in this dissertation and my vision of future work inspires a new generation of researchers and practitioners who can put these ideas into practice, validating, falsifying, adapting, and expanding them to continue advancing the state of computer science education. I also hope that my research stands as proof that, given the appropriate support, ***anyone*** who wants to learn programming can, and that it is our job as computer science education researchers and educators to ensure that our combined efforts make this a reality.

BIBLIOGRAPHY

[1] Vincent Aleven, Ido Roll, Bruce M McLaren, and Kenneth R Koedinger. Automated, unobtrusive, action-by-action assessment of self-regulation during learning with an intelligent tutoring system. *Educational Psychologist*, 45(4):224–233, 2010.

[2] Omar AlZoubi, Davide Fossati, Barbara Di Eugenio, and Nick Green. Chiqat-tutor: An integrated environment for learning recursion. In *Proc. of the Second Workshop on AI-supported Education for Computer Science (AIEDCS). Honolulu, HI*, 2014.

[3] John R Anderson, Frederick G Conrad, and Albert T Corbett. Skill acquisition and the lisp tutor. *Cognitive Science*, 13(4):467–505, 1989.

[4] Petek Askar and David Davenport. An investigation of factors related to self-efficacy for java programming among engineering students. 8(1), 2009.

[5] Richard P Bagozzi, Utpal M Dholakia, and Suman Basuroy. How effortful decisions get enacted: The motivating role of decision processes, desires, and anticipated emotions. *Journal of Behavioral Decision Making*, 16(4):273–295, 2003.

[6] Albert Bandura. Self-efficacy: toward a unifying theory of behavioral change. *Psychological review*, 84(2):191, 1977.

[7] Albert Bandura. Social foundations of thought and action. *Englewood Cliffs, NJ*, 1986:23–28, 1986.

[8] Albert Bandura and Richard H Walters. *Social learning theory*, volume 1. Prentice-hall Englewood Cliffs, NJ, 1977.

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

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

[11] Katerine Bielaczyc, Peter L. Pirolli, and Ann L. Brown. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. 13(2):221–252, 1995.

[12] Lisa S Blackwell, Kali H Trzesniewski, and Carol Sorich Dweck. Implicit theories of intelligence predict achievement across an adolescent transition: A longitudinal study and an intervention. *Child development*, 78(1):246–263, 2007.

[13] Nigel Bosch and Sidney D'Mello. The affective experience of novice computer programmers. *International Journal of Artificial Intelligence in Education*, 27(1):181–206, 2017.

[14] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[15] John D Bransford, Ann L Brown, Rodney R Cocking, et al. *How people learn*, volume 11. Washington, DC: National academy press, 2000.

[16] John D Bransford and Barry S Stein. The ideal problem solver. 1993.

[17] Jill Cao. An idea garden for end-user programmers. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 915–918. 2012.

[18] Jill Cao, Scott D Fleming, and Margaret Burnett. An exploration of design opportunities for "gardening" end-user programmers' ideas. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 35–42. IEEE, 2011.

[19] Jill Cao, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. Idea garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 27(6):640–660, 2015.

[20] Jill Cao, Irwin Kwan, Faezeh Bahmani, Margaret Burnett, Scott D Fleming, Josh Jordahl, Amber Horvath, and Sherry Yang. End-user programmers in trouble: Can the idea garden help them to help themselves? In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 151–158. IEEE, 2013.

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

[22] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, 13(2):145–182, 1989.

[23] Ryan Chmiel and Michael C Loui. *An integrated approach to instruction in debugging computer programs*, volume 3. IEEE, 2003.

[24] Michael J Clancy and Marcia C Linn. *Designing Pascal solutions: A case study approach*. Computer Science Press, Inc., 1992.

[25] Michael J Clancy and Marcia C Linn. *Designing Pascal Solutions: Case studies using data structures*. WH Freeman & Co., 1996.

[26] Douglas H Clements and Dominic F Gullo. Effects of computer programming on young children's cognition. *Journal of Educational psychology*, 76(6):1051, 1984.

[27] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[28] Kent J Crippen and Boyd L Earl. The impact of web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. *Computers & Education*, 49(3):809–821, 2007.

[29] David R Cross and Scott G Paris. Developmental and instructional analyses of children's metacognition and reading comprehension. *Journal of educational psychology*, 80(2):131, 1988.

[30] Carol S Dweck. *Mindset: The new psychology of success*. Random House Digital, Inc., 2008.

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

[32] Mansoor Fahim and Ebrahim Fakhri Alamdari. Maximizing learnersâĂŹ metacognitive awareness in listening through metacognitive instruction: An empirical study. 3(3), 2014.

[33] Katrina Falkner, Claudia Szabo, Rebecca Vivian, and Nickolas Falkner. Evolution of software development strategies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 243–252. IEEE, 2015.

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

[35] Allan Fisher and Jane Margolis. Unlocking the clubhouse: the carnegie mellon experience. *ACM SIGCSE Bulletin*, 34(2):79–83, 2002.

[36] Kathi Fisler. The recurring rainfall problem. In *ACM International Computing Education Conference*, pages 35–42. ACM, 2014.

[37] Kathi Fisler and Francisco Enrique Vicente Castro. Sometimes, rainfall accumulates: Talk-alouds with novice functional programmers. In *ACM International Computing Education Conference*, pages 12–20. ACM, 2017.

[38] Abraham E Flanigan, Markeya S Peteranetz, Duane F Shell, and Leen-Kiat Soh. Exploring changes in computer science students' implicit theories of intelligence across the semester. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, pages 161–168, 2015.

[39] Jamie Gorson and Eleanor O'Rourke. How do students talk about intelligence? an investigation of motivation, self-efficacy, and mindsets in computer science. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 21–29. ACM, 2019.

[40] Nick Green, Omar AlZoubi, Mehrdad Alizadeh, Barbara Di Eugenio, Davide Fossati, and Rachel Harsley. A scalable intelligent tutoring system framework for computer science education. In *International Conference on Computer Supported Education, CSEDU 2015*, 2015.

[41] James G Greeno and Rogers P Hall. Practicing representation: Learning with and about representational forms. *Phi Delta Kappan*, 78:361–367, 1997.

[42] Philip J Guo, Juho Kim, and Rob Rubin. How video production affects student engagement: An empirical study of mooc videos. In *Proceedings of the first ACM conference on Learning@Scale conference*, pages 41–50. ACM, 2014.

[43] David Hammer and Leema K Berland. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. *Journal of the Learning Sciences*, 23(1):37–46, 2014.

[44] Matthias Hauswirth and Andrea Adamoli. Metacognitive calibration when learning to program. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling '17, pages 50–59, New York, NY, USA, 2017. ACM.

[45] Jean-Michel Hoc and Anh Nguyen-Xuan. Language semantics, mental models and analogy. 10:139–156, 1990.

[46] Yun Huang. *Learner modeling for integration Skills in programming*. PhD thesis, University of Pittsburgh, 2018.

[47] Zhexue Huang. A fast clustering algorithm to cluster very large categorical data sets in data mining. *DMKD*, 3(8):34–39, 1997.

[48] Facebook Inc. React: A javascript library for building user interfaces.

[49] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. A study of code design skills in novice programmers using the solo taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 251–259. ACM, 2016.

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

[51] Cem Kaner and Sowmya Padmanabhan. Practice and transfer of learning in the teaching of software testing. In *20th Conference on Software Engineering Education & Training (CSEET'07)*, pages 157–166. IEEE, 2007.

[52] Ada S Kim and Amy J Ko. A pedagogical analysis of online coding tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 321–326. ACM, 2017.

[53] Amy J Ko. Attitudes and self-efficacy in young adults' computing autobiographies. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 67–74. IEEE, 2009.

[54] Amy J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. Teaching explicit programming strategies to adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 469–475. ACM, 2019.

[55] Amy J Ko and Yann Riche. The role of conceptual knowledge in api usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 173–176. IEEE, 2011.

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

[57] Asher Koriat. Metacognition: Decision-making processes in self-monitoring and self-regulation. *The Wiley Blackwell handbook of judgment and decision making*, 1:356–379, 2016.

[58] Michael J Lee, Amy J Ko, and Irwin Kwan. In-game assessments increase novice programmers' engagement and level completion speed. In *Proceedings of the ninth annual*

international ACM conference on International computing education research, pages 153–160. ACM, 2013.

[59] Paul Luo Li, Amy J Ko, and Jiamin Zhu. What makes a great software engineer? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 700–710. IEEE Press, 2015.

[60] Soohyun Nam Liao, Sander Valstar, Kevin Thai, Christine Alvarado, Daniel Zingaro, William G. Griswold, and Leo Porter. Behaviors of higher and lower performing students in cs1. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, pages 196–202, New York, NY, USA, 2019. ACM.

[61] Marcia C Linn and Michael J Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, 1992.

[62] Raymond Lister. Computing education research geek genes and bimodal grades. *ACM Inroads*, 1(3):16–17, 2011.

[63] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, volume 36, pages 119–150. ACM, 2004.

[64] Yanjin Long and Vincent Aleven. Supporting students' self-regulated learning with an open learner model in a linear equation tutor. In *International conference on artificial intelligence in education*, pages 219–228. Springer, 2013.

[65] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112. ACM, 2008.

[66] Jane Margolis. *Stuck in the shallow end: Education, race, and computing*. MIT Press, 2010.

[67] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: Women in computing*. MIT press, 2003.

[68] Santosh A Mathan and Kenneth R Koedinger. Fostering the intelligent novice: Learning from errors with metacognitive tutoring. *Educational psychologist*, 40(4):257–265, 2005.

[69] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. The effects of pair-programming on performance in an introductory programming course. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 38–42, 2002.

[70] Donald Meichenbaum. Cognitive behaviour modification. *Cognitive Behaviour Therapy*, 6(4):185–192, 1977.

[71] Janet Metcalfe, Arthur P Shimamura, et al. *Metacognition: Knowing about knowing*. MIT press, 1994.

[72] Miriam Bassok Michelene T. H. Chi. Self-explanations: How students study and use examples in learning to solve problems. 13(2):145–182, 1989.

[73] Marjorie Montague. Self-regulation strategies to improve mathematical problem solving for students with learning disabilities. *Learning Disability Quarterly*, 31(1):37–44, 2008.

[74] Briana B Morrison, Lauren E Margulieux, Barbara Ericson, and Mark Guzdial. Subgoals help students solve parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 42–47, 2016.

[75] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 21–29, 2015.

[76] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the 11th Annual International Conference on International Computing Education Research*, ICER '15, pages 21–29. ACM, 2015.

[77] Harry G. Murray, J. Philippe, and Sampo V. Paunonen. Teacher personality traits and student instructional ratings in six types of university courses. 82(2):250–261, 1990.

[78] Alok Mysore and Philip J Guo. Torta: Generating mixed-media gui and command-line app tutorials using operating-system-wide activity tracing. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 703–714. ACM, 2017.

[79] Greg L Nelson, Benjamin Xie, and Amy J Ko. 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*, pages 2–11. ACM, 2017.

[80] David B Palumbo and W Michael Reed. The effect of basic programming language instruction on high school studentsâĂŹ problem solving ability and computer anxiety. *Journal of Research on computing in Education*, 23(3):343–372, 1991.

[81] Seymour Papert. Mindstorms: Computers, children, and powerful ideas. *NY: Basic Books*, 1980.

[82] Amy Pavel, Björn Hartmann, and Maneesh Agrawala. Video digests: a browsable, skimmable format for informational lecture videos. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 573–582. ACM, 2014.

[83] Roy D Pea and D Midian Kurland. On the cognitive effects of learning computer programming. *New ideas in psychology*, 2(2):137–168, 1984.

[84] Nancy Pennington and Beatrice Grabowski. The tasks of programming. In *Psychology of programming*, pages 45–62. Elsevier, 1990.

[85] Marian Petre. Insights from expert software design practice. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 233–242. ACM, 2009.

[86] Paul R Pintrich and Elisabeth V De Groot. Motivational and self-regulated learning components of classroom academic performance. *Journal of educational psychology*, 82(1):33, 1990.

[87] Paul R Pintrich, Christopher A Wolters, and Gail P Baxter. 2. assessing metacognition and self-regulated learning. 2000.

[88] Patricia Pokay and Phyllis C Blumenfeld. Predicting achievement early and late in the semester: The role of motivation and use of learning strategies. *Journal of educational psychology*, 82(1):41, 1990.

[89] George Polya. *How to solve it: A new aspect of mathematical method*. Number 246. Princeton university press, 2004.

[90] Leo Porter and Beth Simon. Retaining nearly one-third more majors with a trio of instructional best practices in cs1. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 165–170, 2013.

[91] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 531–537, New York, NY, USA, 2019. ACM.

[92] Martin P Robillard. What makes apis hard to learn? Answers from developers. *IEEE software*, 26(6):27–34, 2009.

[93] Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

[94] Claudia M Roebers. Executive function and metacognition: Towards a unifying framework of cognitive self-regulation. *Developmental Review*, 45:31–51, 2017.

[95] Ido Roll, Vincent Aleven, Bruce M McLaren, and Kenneth R Koedinger. Designing for metacognition–applying cognitive tutor principles to the tutoring of help seeking. *Metacognition and Learning*, 2(2-3):125–140, 2007.

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

[97] D Royce Sadler. Formative assessment and the design of instructional systems. *Instructional science*, 18(2):119–144, 1989.

[98] R Keith Sawyer. The new science of learning. *The Cambridge handbook of the learning sciences*, 1:18, 2006.

[99] Gregory Schraw. Promoting general metacognitive awareness. *Instructional science*, 26(1-2):113–125, 1998.

[100] Gregory Schraw, Kent J Crippen, and Kendall Hartley. Promoting self-regulation in science education: Metacognition as part of a broader perspective on learning. *Research in science education*, 36(1-2):111–139, 2006.

[101] Dale H Schunk and Barry J Zimmerman. Social origins of self-regulatory competence. *Educational psychologist*, 32(4):195–208, 1997.

[102] Michael J Scott and Gheorghita Ghinea. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education*, 57(3):169–174, 2013.

[103] Judy Sheard, S Simon, Margaret Hamilton, and Jan Lönnberg. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 93–104. ACM, 2009.

[104] Robert H Sloan and Patrick Troy. CS 0.5: a better approach to introductory computer science for majors. *ACM SIGCSE Bulletin*, 40(1):271–275, 2008.

[105] Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[106] Elliot Soloway and James C. Spohrer. *Studying the novice programmer*. Psychology Press, 2013.

[107] Elmar Souvignier and Judith Mokhlesgerami. Using self-regulation as a framework for implementing strategy instruction to foster reading comprehension. *Learning and instruction*, 16(1):57–71, 2006.

[108] Rayne A Sperling, Bruce C Howard, Lee Ann Miller, and Cheryl Murphy. Measures of children's knowledge and regulation of cognition. *Contemporary educational psychology*, 27(1):51–79, 2002.

[109] David R Thomas. A general inductive approach for qualitative data analysis. 2003.

[110] Kentaro Toyama. Technology as amplifier in international development. In *Proceedings of the 2011 iConference*, pages 75–82. 2011.

[111] David R Vago and Silbersweig AMD David. Self-awareness, self-regulation, and self-transcendence (s-art): a framework for understanding the neurobiological mechanisms of mindfulness. *Frontiers in human neuroscience*, 6:296, 2012.

[112] April Y Wang, Ryan Mitts, Philip J Guo, and Parmit K Chilana. Mismatch of expectations: How modern learning resources fail conversational programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 511. ACM, 2018.

[113] David Whitebread, Penny Coltman, Deborah Pino Pasternak, Claire Sangster, Valeska Grau, Sue Bingham, Qais Almeqdad, and Demetra Demetriou. The development of two observational tools for assessing metacognition and self-regulated learning in young children. 4(1):63–85, 2009.

[114] Philip H. Winne and Nancy E. Perry. Measuring self-regulated learning. 2000.

[115] John Wrenn and Shriram Krishnamurthi. Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 131–139. ACM, 2019.

[116] Franceska Xhakaj and Vincent Aleven. Towards improving introductory computer programming with an ITS for conceptual learning. In Carolyn Penstein Rosé, Roberto Martínez-Maldonado, H Ulrich Hoppe, Rose Luckin, Manolis Mavrikis, Kaska Porayska-Pomsta, Bruce McLaren, and Benedict du Boulay, editors, *Artificial Intelligence in Education*, volume 10948, pages 535–538. Springer International Publishing, Cham, 2018.

[117] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253, 2019.

[118] Shir Yadid and Eran Yahav. Extracting code from programming tutorial videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 98–111. ACM, 2016.

[119] Barry J Zimmerman. Attaining self-regulation: A social cognitive perspective. In *Handbook of self-regulation*, pages 13–39. Elsevier, 2000.

[120] Barry J Zimmerman. Investigating self-regulation and motivation: Historical background, methodological developments, and future prospects. *American educational research journal*, 45(1):166–183, 2008.