Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance

Dastyni Loksa[1], Amy J Ko[1], Will Jernigan[2], Alannah Oleson[2], Christopher J Mendez[2], and Margaret M Burnett[2]

The Information School, University of Washington DUB Group Seattle, Washington {dloksa, ajko}@uw.edu

School of EECS, Oregon State University Corvallis, Oregon {jernigaw, olesona, mendezc, burnett}@oregonstate.edu

ABSTRACT

More people are learning to code than ever, but most learning opportunities do not explicitly teach the problem solving skills necessary to succeed at open-ended programming problems. In this paper, we present a new approach to impart these skills, consisting of: 1) explicit instruction on programming problem solving, which frames coding as a process of translating mental representations of problems and solutions into source code, 2) a method of visualizing and monitoring progression through six problem solving stages, 3) explicit, on-demand prompts for learners to reflect on their strategies when seeking help from instructors, and 4) context-sensitive help embedded in a code editor that reinforces the problem solving instruction. We experimentally evaluated the effects of our intervention across two 2-week web development summer camps with 48 high school students, finding that the intervention increased productivity, independence, programming self-efficacy, metacognitive awareness, and growth mindset. We discuss the implications of these results on learning technologies and classroom instruction.

Author Keywords
Problem-solving; Programming; Metacognition; Computer Science Education

ACM Classification Keywords
K.3.2 Computer Science Education; H.5.m Information interfaces and presentation (e.g., HCI): Miscellaneous

INTRODUCTION

Programming is rapidly becoming a 21st century literacy [45], with demand for software developers in the U.S. alone projected to grow at twice the rate of the average occupation through 2022 [8]. With this rise in demand for coding skills, there has also been a rise in the desire to learn to code [14], with millions using online sites such as *code.org*, *codecademy.org*, and *khanacademy.org* and tens of thousands enrolling in coding boot camps and CS programs. Countries around the world are even beginning to require coding classes in K-12 curricula, causing teachers to search for both learning technologies that teach coding and techniques for teaching with them effectively. The trend is clear: the ability to use programming languages—the most powerful of human-computer interfaces—is *the* skill to learn.

Although the availability of opportunities to learn to code is now very high, evidence suggests that these opportunities leave much room for improvement. Students continue to drop out of introductory programming courses at rates of 30-50% [5], often because they find the material too difficult [20]. Online tutorials such as *codecademy.org* and massively open online courses, while boasting millions of users, have attrition rates as high as 90% [27]. Even when learners complete these courses, they still score poorly on tests of basic coding knowledge [33]. Worse yet, recent work has found that introductory CS courses can convince learners' that their abilities are fixed and cannot be improved with practice [21,49], deterring them from not only learning to code, but learning any new skill.

There are many technologies designed to teach coding in more engaging ways, including the widely disseminated Scratch [41] and Alice [28]. However, studies of these learning technologies show that rather than use them to learn to code, most learners primarily use them to create content, possibly avoiding coding altogether [46]. Even with educational games such as Gidget [32], which are explicitly designed to engage learners in coding, the *best* that students who complete these games do on validated tests of programming knowledge is only 50% [33].

Some of these poor learning outcomes are due to social factors such as stereotype-reinforcing student behavior [20,36,51], teacher bias against students without "the geek gene" [35], and learners' lack of intrinsic interest in computing [20]. However, research shows that *how* coding is taught is also important. For instance, there are several

ACM 978-1-4503-3362-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2858036.2858252

evidence-based instructional techniques that can substantially improve learning and reduce attrition [40]. There is also evidence that without these techniques, many learners struggle through courses, feeling disoriented, lost, frustrated, and unsupported [30].

How can online tutorials like *codecademy.org*, creative technologies like Scratch [41], and traditional classroom environments improve these outcomes? In this paper, we investigate the idea that we should go beyond teaching just programming languages and tools, to also teaching the *cognitive* aspects of programming. After all, coding involves skills that go well beyond how to use a language. For instance, a recent study investigating software engineering expertise found that great engineers are *systematic* and *self-aware* [34]. Similarly, 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 [24,19,42,49]. There is also evidence that the highest performing CS students are the ones who are most aware of their problem solving state and most capable of managing their cognitive resources [6]. This suggests that learning to code not only requires effective instruction on syntax, data structures, and abstraction, but also the development of metacognitive awareness [37].

Unfortunately, there is little insight in prior work about how to promote metacognitive awareness in programming. The closest and most recent effort is the *Idea Garden* [13], 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 some evidence that contextual hints help learners succeed more independently [26] and that scaffolding metacognitive work is beneficial in invention activities [44], but these are not designed to promote metacognitive awareness. Aside from these, most of the work concerning programming and metacognition claims that programming *develops* stronger general metacognitive awareness [15,39], but this work is both disputed [4] and says nothing about how to teach metacognitive awareness in programming.

In this paper, we contribute an approach to promoting metacognitive awareness in introductory programming settings and investigate its effects on help requests, productivity, self-efficacy, and growth mindset. Our approach is grounded in the idea that programming is not merely about language syntax and semantics, but more fundamentally about the *iterative process of refining mental representations of computational problems and solutions* and expressing those representations as code. We use this framing of programming in an integrated set of four interventions: 1) an interactive lecture on problem solving in programming, 2) a physical model of problem solving stages that learners can use to track their problem solving state, 3) explicit prompts for learners to describe their problem solving state when they request help, and 4) context-sensitive problem solving hints in an IDE.

To explore and evaluate the impact of these metacognitive interventions (our first contribution), we conducted a controlled experiment (our second contribution) 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 [2,3]), foster growth mindsets (their theories about whether ability is learned or innate [18]), and ultimately produce a higher quantity of functional code. In the rest of this paper, we describe our intervention, 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.

THE APPROACH: PROBLEM SOLVING STAGES AND METACOGNITIVE PROMPTS

Our approach to teaching problem solving in programming draws upon work on problem solving, metacognition, and the psychology of programming. We derive our approach from 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 [43]. In programming, we believe this means: 1) providing knowledge of the range of activities that programmers engage in to solve problems, 2) imparting ways that programmers converge toward a solution, and 3) teaching to reflect on and regulate strategies.

There are many techniques in prior literature for teaching such metacognitive skills. For example, in prior work on problem solving, studies have taught learners about general limitations and biases in human learning and memory [16,48] and provided planning, monitoring, and evaluation checklists in reading and math [47]. These can have positive benefits on learning outcomes, self-efficacy, and growth mindset [22,17,38,52].

In our work, we propose two interventions that 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.

The problem solving stages we propose to teach include six stages that prior literature on the psychology of programming suggests are essential to successful programming. While nominally sequential, the stages are re-visited frequently as programmers iteratively implement a solution and discover knowledge about the problem and solution that was not initially apparent. The stages 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" [23]. The more explicit this interpretation process, the more likely a programmer will overcome ambiguities in the problem [42].
- *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) [25]. By reusing knowledge of related problems, programmers can better conceptualize a problem's computational nuances.
- *Search for 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 [9,29].
- *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 [31].
- *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 [29,42].

One can instantiate instruction on these six stages 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. In the next section, we describe how we evaluated our particular instantiation of these interventions, providing one example of how they might work in practice.

METHODS

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 we described in the previous section. In this section, we describe our two camps and the data we collected to measure the effects of our intervention.

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 their non-primary 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 contained similar numbers of females (X^2=0, df=1, p=1.000), they had similar grade levels (X^2=4.1829, df=3, p=0.242), and similar self-reported programming and web development experience (X^2 = 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 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://*c9.io*).

The Instruction

We aimed to teach concepts, syntax, and semantics of HTML, CSS, and JavaScript with a focus on the *React*

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; Problem solving reminder *(experimental only)*
Days 5-9	Free development time
Day 10	Project presentations

Table 1: The camp schedule, with experimental camp's additions as noted.

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 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 (the first author), presented HTML, JavaScript, and React lectures to both groups. Another instructor (the third author) presented a CSS lecture and a growth mindset exercise to both groups. Three additional undergrads also acted as helpers. All members of the instructional team had at least novice experience with web development. The lead instructor had no experience running camps or teaching programming.

The 1-hour problem solving lecture (the first part of our intervention, given only to the experimental group) taught campers the six programming problem solving stages we described earlier. The instructor began the lecture with a book sorting exercise. He asked the campers how to sort the books by size and followed their verbal instructions. Next, he asked 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 stages, starting with *reinterpreting the problem prompt*. Campers tried to identify the next stage of the process in groups at the instructor's request. Once the campers identified the next stage (or the instructor identified it when campers ran out of ideas), he tied abstract concept of the stage to a concrete problem, such as the book sorting problem the lecture began with.

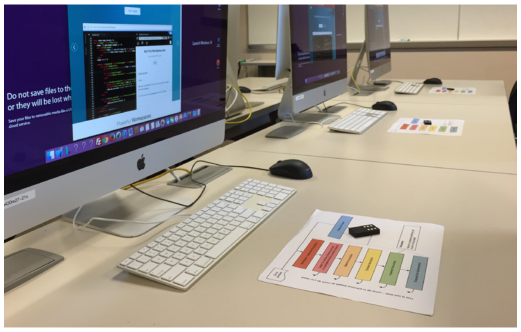


Figure 1: The paper handout and physical token we gave to campers to track their problem solving stage.

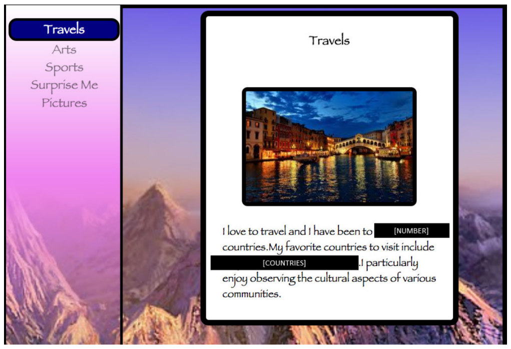
After the lecture, we provided the experimental group with a physical handout of the problem solving stages (shown in Figure 1) and a physical token so they could track their current state on the handout (the second part of our intervention). We instructed campers to track their progress through the stages 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 stages, it did not prescribe *how* they achieve it. We did not mention any particular strategies or resources to use for each stage. 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 also told the campers they could use the Idea Garden, which mentions some strategies such as *working backwards*.

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 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 2).

During both the after-lecture activities and project work time, campers in both groups had access to several types of



I love to travel and I have been to [NUMBER] countries.My favorite countries to visit include [COUNTRIES].I particularly enjoy observing the cultural aspects of various communities.

Figure 2: Camper E27's final project, showing buttons that link to different interests (left) and content and images (center). Details have been anonymized.

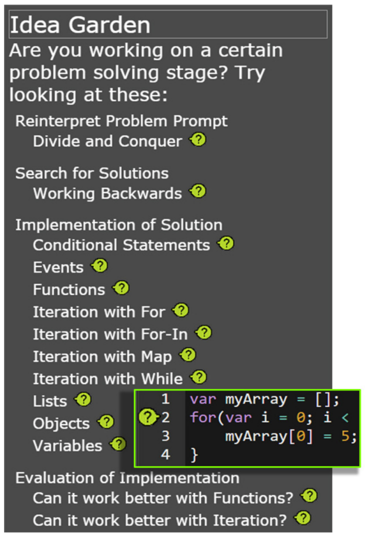
Task	Content	HTML	CSS	JS
Add a window title to the web page		✓		
Create objects to represent each of your interests	✓			✓
Change the background color and add a border to your page			✓	
Create a space for each of your interest's names		✓		✓
Add a component that displays a photo of your interest				✓
Display 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 2: Condensed versions of the prescribed tasks given to the campers and the skills that each task required.

help. We gave campers 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 to do so, 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 stage do you think you are in?* (The third part of our intervention). 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 practice this process, the helpers trained in a 3-day pilot camp.

To provide context-sensitive problem solving prompts to the experimental group (the fourth part of our intervention), we implemented the Idea Garden [10, 11, 12, 13, 26] in a panel of the Cloud9 IDE (see Figure 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



Idea Garden
Are you working on a certain problem solving stage? Try looking at these:
Reinterpret Problem Prompt
Divide and Conquer
Search for Solutions
Working Backwards
Implementation of Solution
Conditional Statements
Events
Functions
Iteration with For
Iteration with For-In
Iteration with Map
Iteration with While
Lists
Objects
Variables
Evaluation of Implementation
Can it work better with Functions?
Can it work better with Iteration?

```
var myArray = [];
for(var i = 0; i <
    myArray[0] = 5;
}
```

Figure 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.

reinforced the problem solving stages by housing the Idea Garden's 14 hints under headers corresponding to the six stages. 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 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 [55, 53], 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. [1] to fit 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 [49]. 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.

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 1, 3) the help request prompts, and 4) the Idea Garden help shown in Figure 3. In this section, we describe the changes these additions caused, beginning with a qualitative description of the campers' experiences and outcomes to give context to our results. We 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.

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 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 map hint] 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.

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 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 on 20% of the data). 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

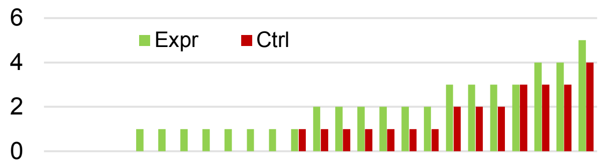


Figure 4 (above): The total number of strategies mentioned in end-of-day survey responses by campers in each group, sorted by frequency. The experimental group mentioned more strategies than the control group.

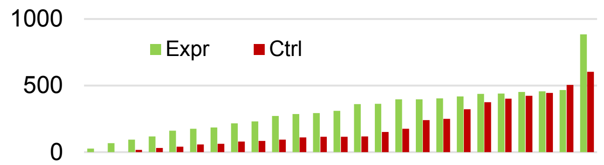


Figure 5 (above): The total word count of all end-of-day survey responses by campers in each group, sorted by count. The experimental group wrote more than the control group.

Figure 4). As shown in Figure 5, the experimental group campers also wrote significantly more words in their responses (H=6.326, p=0.011).

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 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 [29]. We list the six barriers in Table 3, showing examples from campers. Each barrier is a general type of impasse that learners typically encounter in programming tasks. Table 4 lists some of the barriers that might occur in particular problem solving stages.

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 3, the proportion of help request types varied significantly by condition (X^2=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, it appears that campers in the experimental group were more likely to select a solution and implement it independently, allowing them to progress to evaluation before requiring help.

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 4: The barriers from [29] that might be encountered in a particular problem solving stage.

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 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.

Barrier	Definition from [29]	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%
Understanding	Observed a failure and did not have guesses about why it was failing.	*"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 it.	*"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 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.

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 according to which skills they required (as indicated in Table 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, 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 7). Figure 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 9).

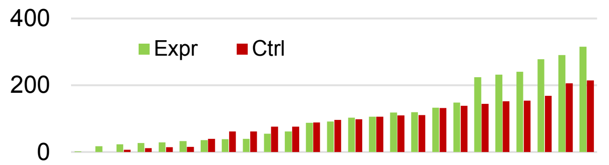


Figure 6 (above): Campers' prescribed task productivity scores by condition, sorted in increasing order. The experimental campers' productivities were typically about equivalent to or higher than the control campers'.

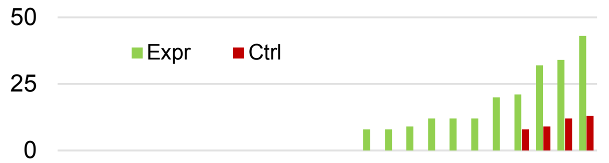


Figure 7 (above): Campers' self-initiated task productivity scores by condition, sorted in increasing order. Experimental campers' productivities were significantly higher than control campers'. Values of zero are not visible.

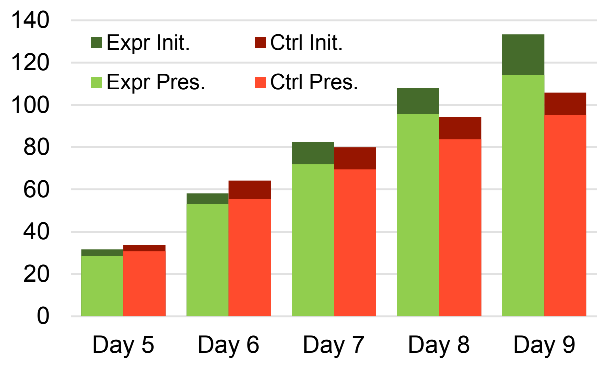


Figure 8 (above): Cumulative average productivities per project day on both prescribed (light hues) and self-initiated (dark hues) tasks. The experimental group was increasingly more productive than the control group.

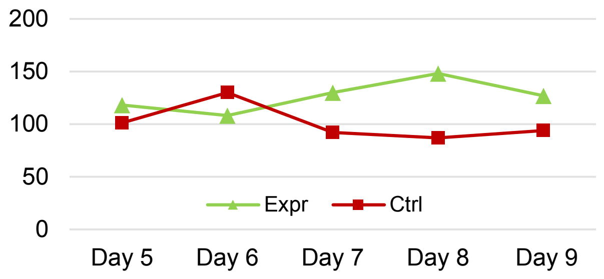


Figure 9: The campers' median lines of code changes per project day by condition. Experimental campers' amount of code changed was not significantly different from control campers' except on day 8.

This may suggest 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.

Impact on Self-Efficacy

With the experimental group's greater productivity, we also expected to see a relative increase in self-efficacy between the two groups. 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 10 shows the distributions of these scores before and after the camp by condition, and Figure 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. 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 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 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.

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

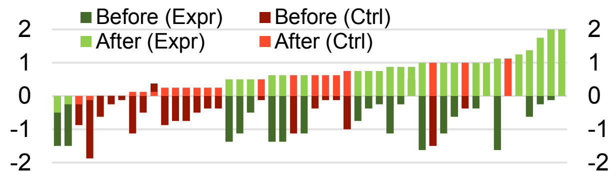


Figure 10: Aggregate self-efficacy scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Self-efficacy increased for most campers, but increased significantly more in the experimental group.

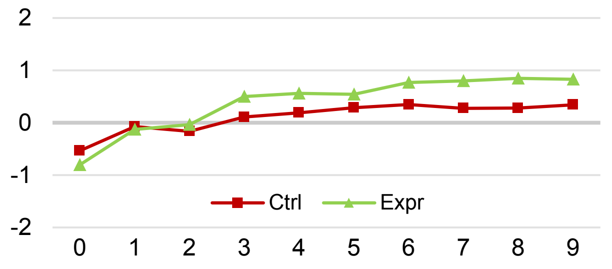


Figure 11: Mean self-efficacy in each group for each day of the camp. The experimental campers' self-efficacy increased after the introduction of the intervention (day 2) and ended mildly positive, while the control campers' ended at a neutral level.

[21]. We hypothesized that by increasing campers' success at problem solving and programming, we could prevent this erosion of growth mindset. To test this hypothesis, we mapped 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 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 (slightly below "agree" on our scale), (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 [21]), 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.

DISCUSSION AND FUTURE WORK

Our results provide some of the first evidence that teaching problem solving for programming is not only possible, but

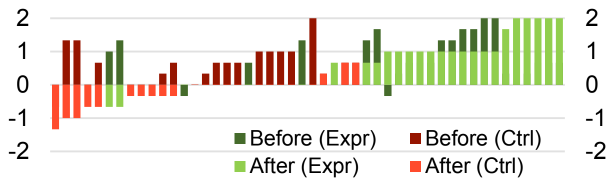


Figure 12: Aggregate growth mindset scores for all campers in both groups, with experimental and control sorted from lowest to highest. Some before values are not visible due to high after values. Growth mindset stayed positive in the experimental group but turned more negative in the control.

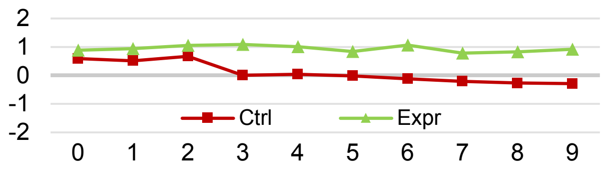


Figure 13: Mean growth mindset score across the ten days, by condition. While the control campers' growth mindsets deteriorated sharply when JavaScript was introduced (day 3) and continued to degrade, the experimental campers maintained their existing growth mindset throughout the camp and showed a slight upward trend.

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 [43]—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" coding, 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 [7] and the tendency of introductory programming settings to weaken it [21,49], 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 programing 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 [20].

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. Future work is likely to find that how problem solving is taught is just as important as teaching it at all.

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 [54], 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.

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

REFERENCES

1. Petek Askar and David Davenport. 2009. An investigation of factors related to self-efficacy for Java programming among engineering students. *Online Submission* 8, 1. http://eric.ed.gov/?id=ED503900.

2. Albert Bandura. 1977. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review 8*, 2: 191-215.

3. Albert Bandura. 1986. *Social Foundations of Thought and Action*. Prentice Hall, Englewood Cliffs, NJ.

4. Henry Jay Becker. 1987. The importance of a methodology that maximizes falsifiability: Its applicability to research about Logo. *Educational Researcher*, 16, 5: 11-16.

5. Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39, 2: 32-36.

6. Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the International Workshop on Computing Education Research (ICER '05)*, 81-86.

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

8. Bureau of Labor Statistics. U.S. Department of Labor, Occupational Outlook Handbook, 2014-15 Edition, Software Developers. Retrieved August 11, 2015 from http://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm.

9. Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*, 1589-1598.

10. Jill Cao, Scott D. Fleming, and Margaret Burnett. 2011. An exploration of design opportunities for "gardening" end-user programmers' ideas. In *Proceedings of the IEEE symposium on Visual Languages and Human-Centric Computing (VL/HCC '11)*, 35-42.

11. Jill Cao. 2012. An idea garden for end-user programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Extended Abstracts on Human Factors in Computing Systems* (*CHI EA '12*), 915-918.

12. Jill Cao, Irwin Kwan, Faezeh Bahmani, Margaret Burnett, Scott D. Fleming, Josh Jordahl, Amber Horvath, and Sherry Yang. 2013. End-user programmers in trouble: Can the Idea Garden help them to help themselves? In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (*VL/HCC '13*), 151-158.

13. Jill Cao, Scott D. Fleming, Margaret Burnett, and Christopher Scaffidi. 2014. Idea Garden: situated support for problem solving by end-user programmers. *Interacting with Computers*.

14. Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J. Guo. 2015. Perceptions of Non-CS Majors in Intro Programming: The Rise of the Conversational Programmer. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, to appear.

15. Douglas H. Clements and Dominic F. Gullo. 1984. Effects of computer programming on young children's cognition. *Journal of Educational Psychology* 76, 6: 1051.

16. David R. Cross and Scott G. Paris. 1988. Developmental and instructional analyses of children's metacognition and reading comprehension. *Journal of Educational Psychology* 80, 2: 131.

17. Savia Coutinho. 2008. Self-efficacy, metacognition, and performance. *North American Journal of Psychology* 10, 1: 165.

18. Carol Dweck. 2006. *Mindset: The new psychology of success*. Random House.

19. Anneli Eteläpelto. 1993. Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research* 37, 3: 243-254.

20. Allan Fisher and Jane Margolis. 2002. Unlocking the clubhouse: the Carnegie Mellon experience. *ACM SIGCSE Bulletin* 34, 2: 79-83.

21. Abraham E. Flanigan, Markeya S. Peteranetz, Duane F. Shell, and Leen-Kiat Soh. 2015. Exploring changes in computer science students' implicit theories of intelligence across the semester. In *Proceedings of the International Conference on Computing Education Research (ICER '15)*, 161-168.

22. Forrest-Pressley, Donna-Lynn, and G. E. MacKinnon. 1985. *Metacognition, Cognition, and Human Performance: Theoretical perspectives*. Vol. 1. Academic Press.

23. James G. Greeno and Rogers P. Hall. 1997. Practicing representation. *Phi Delta Kappan* 78, 5: 361.

24. M. Havenga, 2011. Problem-solving processes in computer programming: a case study." In *Southern African Computer Lecturers' Association (SACLA) Conference Proceedings*, 91-99.

25. Jean-Michel Hoc and Anh Nguyen-Xuan. 1990. Language semantics, mental models and analogy. *Psychology of programming*, 10: 139-156.

26. William Jernigan, Amber Horvath, Michael J. Lee, Margaret M. Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, Amy J. Ko. 2015. A principled evaluation for a principled Idea Garden. In *Proceedings IEEE Visual Languages and Human-Centric Computing (VL/HCC '15)*, to appear.

27. Katy Jordan. 2014. Initial trends in enrollment and completion of massive open online courses. *The International Review Of Research In Open And Distributed Learning* 15, 1.

28. Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1455-1464.

29. Amy J. Ko, Brad Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems." In the *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing,* 199-206.

30. Amy J. Ko. 2009. Attitudes and self-efficacy in young adults' computing autobiographies. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '09),* 67-74.

31. Amy J. Ko and Yann Riche. 2011. The role of conceptual knowledge in API usability. In the *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),* 173-176.

32. Michael J. Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, Sheridan Long, Margaret M. Burnett, and Amy J. Ko. 2014. Principles of a debugging-first puzzle game for computing education. In the *IEEE Symposium* on *Visual Languages and Human-Centric Computing (VL/HCC '14),* 57-64.

33. Michael J. Lee and Amy J. Ko. 2015. Comparing the effectiveness of online learning approaches on CS1 learning outcomes. In *Proceedings of the International Conference on Computing Education Research (ICER '15)*, 237-246.

34. Paul Luo Li, Amy J. Ko, and Jiamin Zhu. 2015. What makes a great software engineer? In *Proceedings of the International Conference on Software Engineering (ICSE '15),* 700-710.

35. Raymond Lister. 2011. Geek genes and bimodal grades. *ACM Inroads* 1, 3: 16-17.

36. Charlie McDowell, Linda Werner, Heather Bullock, Julian Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin*, 34, 1: 38-42.

37. Janet Ed Metcalfe and Arthur P. Shimamura. 1994. *Metacognition: Knowing about knowing*. The MIT Press.

38. John L. Nietfeld and Gregory Schraw. 2002. The effect of knowledge and strategy training on monitoring accuracy. *The Journal of Educational Research* 95, 3: 131-142.

39. David B. Palumbo and W. Michael Reed. 1991. The effect of BASIC programming language instruction on high school students' problem solving ability and computer anxiety. *Journal of Research on computing in Education* 23, 3: 343-372.

40. Leo Porter and Beth Simon. 2013. Retaining nearly one-third more majors with a trio of instructional best practices in CS1. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, 165-170.

41. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Bren.nan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Communications of the ACM* 52, 11: 60-67.

42. Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering,* 30, 12: 889-903.

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

44. Ido Roll, Natasha G. Holmes, James Day, and Doug Bonn. Evaluating Metacognitive Scaffolding in Guided Invention Activities. *Instructional Science* 40, no. 4 (2012): 691–710.

45. Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (*VL/HCC '05*), 207-214.

46. Chris Scaffidi and Chris Chambers. 2012. Skill progression demonstrated by users in the Scratch animation environment. *International Journal of Human-Computer Interaction*, 28, 6: 383-398.

47. Gregory Schraw. Promoting general metacognitive awareness. *Instructional science* 26, 1-2: 113-125.

48. Gregory Schraw, Kent J. Crippen, and Kendall Hartley. 2006. Promoting self-regulation in science education: Metacognition as part of a broader perspective on learning. *Research in Science Education* 36, 1-2: 111-139.

49. Michael James Scott and Gheorghita Ghinea. 2014. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education*, 57, 3: 169-174.

50. Teresa M. Shaft. 1995. Helping programmers understand computer programs: the use of metacognition. *ACM SIGMIS Database* 26, 4: 25-46.

51. Robert H. Sloan and Patrick Troy. 2008. CS 0.5: a better approach to introductory computer science for majors. *ACM SIGCSE Bulletin*, 40, 1: 271-275.

52. Abdrabo Moghazy Soliman and Elsayed Khaled Mathna. 2009. Metacognitive strategy training improves driving situation awareness. *Social Behavior and Personality: An International Journal* 37, 9: 1161-1170.

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

54. Kentaro Toyama. 2011. Technology as amplifier in international development. In *Proceedings of the iConference*, 75-82.

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