Teaching Explicit Programming Strategies to Adolescents

Andrew J. Ko
University of Washington
Information School
Seattle, WA, USA
ajko@uw.edu

Thomas D. LaToza
Department of Computer Science,
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

Stephen Hull
Department of Computer Science,
George Mason University
Fairfax, VA, USA
shull4@gmu.edu

Ellen A. Ko
Juanita High School
Kirkland, WA, USA
s-eko@lwsd.org

William Kwok
University of Washington
Information School
Seattle, WA, USA
wkwok16@uw.edu

Jane Quichocho
University of Washington
Information School
Seattle, WA, USA
janeq97@uw.edu

Harshitha Akkaraju
University of Washington
Information School
Seattle, WA, USA
akkarh@uw.edu

Rishin Pandit
Thomas Jefferson High School for
Science & Technology
Alexandria, VA, USA
rishin.pandit@gmail.com

ABSTRACT

One way to teach programming problem solving is to teach explicit, step-by-step strategies. While prior work has shown these to be effective in controlled settings, there has been little work investigating their efficacy in classrooms. We conducted a 5-week case study with 17 students aged 15-18, investigating students' sentiments toward two strategies for debugging and code reuse, students' use of scaffolding to execute these strategies, and associations between students' strategy use and their success at independently writing programs in class. We found that while students reported the strategies to be valuable, many had trouble regulating their choice of strategies, defaulting to ineffective trial and error, even when they knew systematic strategies would be more effective. Students that embraced the debugging strategy completed more features in a game development project, but this association was mediated by other factors, such as reliance on help, strategy self-efficacy, and mastery of the programming language used in the class. These results suggest that teaching of strategies may require more explicit instruction on strategy selection and self-regulation.

ACM Reference Format:
Andrew J. Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching Explicit Programming Strategies to Adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287371

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02....$15.00.
https://doi.org/10.1145/3287324.3287371

1 INTRODUCTION

Programming is hard to learn [21]. It requires the mastery of programming language semantics [18], common patterns of computation [15], ever-changing APIs and tools [13], and several software engineering skills, such as testing, debugging, and program design [12]. Learners also need strong self-regulation skills, both to regulate their learning [7], but also to regulate their programming process [16, 17]. The range of skills required in programming may be one reason why teaching programming is so difficult [11, 21].

One approach to improving learning is to teach *strategic* skills [19]. For example, one strategic skillset is self-regulation, helping students to reflect on and change their strategies when they find them to be ineffective. Studies show that learners' self-regulated learning skills tend to be shallow, but when strong, are associated with learning success [10, 16, 20]. Others have investigated the self-regulated learning strategies that higher education CS students find effective [7], including explicitly assessing task difficulty. One study developed adolescents' self-regulation skills by prompting learners to reflect on their strategies; this increased students' independence, sustained their growth mindset, and increased their productivity [17], mirroring findings in science education (e.g., [1]).

While self-regulation appears to be key in helping learners select strategies, a related approach has been to teach explicit, step-by-step strategies for solving specific problems. For example, a strategy for debugging might involve prompting students to first find strong evidence of the cause of a failure, and only then edit their program to repair the defect. Experiments on explicit strategies for program design [8, 22], program tracing [23], code reviews [5], and spreadsheet modification [3] have all showed promising short term gains in adult problem solving. Our recent work has further shown that explicit strategies supported by a tool for managing the execution of strategy steps [14] cause experienced developers to have greater task success and more confidence in their progress [14].

While this prior work establishes that self-regulation skills may be important and that explicit strategies for specific programming

tasks can help in controlled settings with adults, there is little prior work investigating the teaching of explicit programming strategies in classrooms. Moreover, even less work has considered teaching strategies to adolescents, who may have distinct challenges from students in higher education, due to their still-developing executive functioning [2]. These gaps in prior work leave open several important questions about the efficacy of strategies in classrooms:

- RQ1: *What barriers do students face trying to use explicit programming strategies in their problem solving?* For example, from an adoption perspective, do students welcome the guidance that explicit strategies can offer, or do they view them too structured and time-consuming?
- RQ2: *To what extent is scaffolding necessary to support the execution explicit strategies?* Can strategy support eventually be removed or is it augmentation that remains valuable even after having learned a strategy?
- RQ3: *To what extent is using explicit strategies associated with more success at programming problem solving?* And to what extent do other factors such as prior knowledge and self-efficacy mediate this success?

To answer these questions, we conducted a classroom study of explicit programming strategies, teaching a 5-week summer course to a group of 17 adolescents. We taught students the game design subset of Code.org's CS Discoveries curriculum and two explicit programming strategies for debugging and code reuse. In the rest of this paper, we detail the course, the strategies, our answers to the questions above, and their implications.

2 METHOD

Our approach to teaching explicit strategies was to offer deliberate practice [6], including direct instruction on two strategies, concrete contextual guidance on using the strategies, and feedback on their use throughout the duration of our. In the rest of this section, we detail the classroom context of this strategy practice, the strategies we taught, and the data we gathered.

2.1 Setting and Participants

We focused on students aged 14-18 in U.S. high schools who were novice to programming. To reach diverse students, we partnered with a university's Upward Bound (UB) program. UB is a U.S. federally-funded college preparation program that helps students who are low-income and/or have no parent or guardian with a bachelor's degree access higher education. The program we worked with served four public high schools and reached about 180 students per year. The program was free; students received lunch money to attend and a stipend upon completion.

UB's summer session last 5-weeks. We offered a "Game Design" elective. The UB staff solicited students' elective preferences and then randomly assigned students to their 1st and 2nd choices. After enrollment stabilized, we had 17 students. The group was mostly low-income racial minority students with little to no programming experience. Students were aged 14-17 years old. Of the 17 students, 9 identified as boys and 8 as girls. All but one student identified as Asian, African American, Hispanic, and Middle Eastern. About 65% of students reported speaking a language other than English fluently, as well as 71% speaking a non-English language at home. The languages reported included English (15), Vietnamese (3), Bengali (2), Somali (2), Cham (2), Mien (1), Russian (1), Nepali (1), and Amharic (1). Students' parents' education mostly ranged between completing some high school to some college.

Students had little to no prior experience with programming: 12 reported never having written a program, but 9 reported having used at least one programming language, including HTML (4), Minecraft (3), Excel (3), Scratch (2). Three students mentioned professional languages such as Java (2), Python (2), and JavaScript (1). To assess prior programming knowledge, we gave students a pre-test which measured their knowledge of basic programming principles such as variables, conditionals, loops, and Game Lab APIs, all in JavaScript. Out of 10 points on the pre-test, the median was 3. One outlier student correctly answered 8 questions.

2.2 Course design

Our course spanned 18 contact hours across 18 days. Class was in a computer lab that seated 25. The 1st author taught the class with the help of four teaching assistants (three undergrads and one high school student). The class followed Code.org's game design curriculum, which used Game Lab, a simple web-based IDE with both a block-based and text-based editor for authoring 2D interactive games with JavaScript. The Code.org curriculum spans more than 30 hours of instruction, so we excluded some lessons. The first 3 weeks of the course covered the subset of JavaScript used in Game Lab and key Game Lab APIs on sprites, animations, and collision detection. Each class began with a brief 5-10 minutes of direct instruction on the lesson for the day, followed by a 45 minute period of self-guided Code.org instruction.

After 3 weeks of instruction, we administered a midterm, then began a 2-week period of game development in which students worked alone or in pairs to design and implement a simple game of their own design across ten class periods. To receive full credit, students' projects needed to: 1) have a background, 2) have a sprite controlled by the player, 3) have a sprite that moves automatically, 4) draw at least one shape, 5) have an animated sprite, and 6) have at least one sprite that responded to collisions. We offered students extra credit for features they wrote down and implemented beyond the basic requirements, incentivizing independent work.

2.3 Explicit strategies

We taught two explicit strategies during the class. To represent strategies, we used a format called Roboto [14], and a tool that helps students execute each step of a Roboto strategy while tracking their progress. Figures 1 and 2 show the text of the strategies and Figure 3 shows the strategy tracker.

The *debug* strategy, shown in Figure 1, was a generic approach to localizing a defect by brainstorming possible causes and investigating each one systematically. If this approach failed, the strategy prompted students to ask the teacher for ideas on possible causes. We taught the strategy just after the lesson that introduced conditionals. To teach the strategy, we discussed the metaphor of fixing a car engine. We asked students if they would use a strategy of unscrewing something without first understanding how the engine worked; most agreed that would be a bad strategy. We then discussed how debugging programs was they same, requiring one to

```
# If you need help finding the problem, ask for help.
Find what your program is doing that you do not want it to do
# Write the line number inside of the program
# and separate with commas.
SET 'possibleCauses' to any lines of the program that
    might be responsible for causing that incorrect 'behavior'
FOR EACH 'cause' IN 'possibleCauses'
    Navigate to 'cause'
    # Ask for help if you need guidance on how.
    Look at the code to verify if it causes the incorrect behavior
    IF 'cause' is the cause of the problem
        # If you need help finding the problem, ask for help.
        Find a way to stop 'cause' from happening
        # Ask for help if you need guidance on how.
        Change the program to stop the incorrect behavior
        # Ask for help if you need guidance on how.
        Mark the task as finished
        RETURN nothing
IF you did not find the cause
    Ask for help finding other possible causes
    Restart the strategy
RETURN nothing
```

Figure 1: The debug strategy.

```
# Describe the behavior as specifically as possible.
SET 'behavior' TO the behavior youd like 'program' to exhibit
IF I know how to implement 'behavior'
    Write code for the 'behavior'
    Run 'program' to test if it exhibits 'behavior' as you expect
    Mark this task as finished
    RETURN nothing
# Use a search query that contains the language you are using and
# describes 'behavior'
Using a search engine such as Google search for relevant examples
UNTIL you have looked at ALL relevant examples
    IF 'result' example seems to be related to your goal
        Click on the link and read the text
        IF the example seems like it might help
            Copy and paste the code into your code editor and ask a
                teacher for help if you don't understand it
            # Change any values to match your needs
            Adapt the code to your needs deleting any lines that you
                are confident are not necessary
            Test your program to ensure the behavior works
            IF the program does not work
                Use the debug strategy to find out why
                RETURN nothing
IF you have not yet found a example
    Ask your teacher for guidance
```

Figure 2: The reuse strategy.

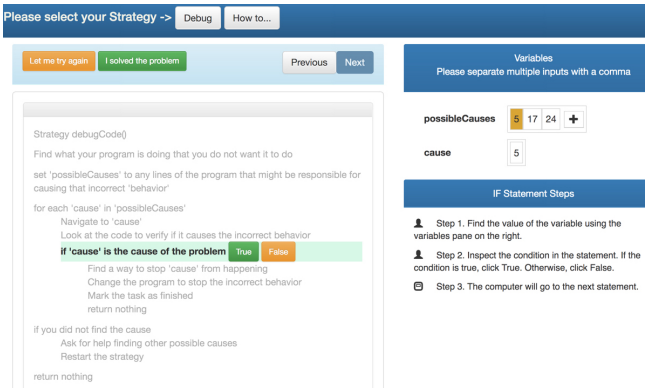


Figure 3: The strategy tracker, showing a conditional step of the debug strategy. The tracker acts as a scratchpad for relevant information and enforces the steps of the strategy.

first understand how the program works and only then edit it. We walked through an example of using the strategy. Then, for the next week, when students asked for help debugging, we prompted them to use the debugging strategy, modeling each step for them, and showing how the strategy could structure their process.

Just before students began developing their own games, we taught the *reuse* strategy shown in Figure 2. The goal of this strategy was to guide program design, helping students to identify the behavior they wanted to implement, then utilize resources such as the web, documentation, and instructors to identify an example to adapt to their game. (To students, we called this the "how to" strategy, but we will call it the *reuse* strategy in this paper). To teach this strategy, we provided direct instruction about how to use it, walked the class through an example, and then modeled its use during one-on-one help requests by showing students how to execute each step of it.

When students had questions in class, we solicited information about their strategy use as follows: 1) we asked "What is your goal right now?", 2) then asked "What strategy are you using to achieve the goal?", and 3) then provided help. While this disincentivized independent use of strategies, it was an opportunity to provide individualized instruction on the strategies, as well as gather data about strategy use. Additionally, not providing help would have subverted the learning goals of the class, which took priority over the research goals. Because of this, we did not expect all students to have a strategy or be using a strategy that we taught. When a student did report using a strategy, it was evident, because they either referred explicitly to a strategy we had taught or struggled to articulate a strategy, instead describing actions they had taken.

One confounding factor in the study was that students needed to feel safe asking for help, otherwise they would not see the strategies modeled, or learn effectively. To mitigate this, we focused on developing trust and rapport with the students, sharing personal details about our experiences with games, game development, programming, and college. All instructors learned students' names and worked with them one-on-one.

2.4 Data collection

To measure barriers to using strategies (RQ1), we gathered three forms of data. The first was during the help requests described above, in which we gathered students' reactions to being taught the strategy (e.g., reluctance, eager opening of the tracker), what help we provided, the outcome of providing the help, and students' reaction to the help we provided. A second source was a daily survey at the end of each class involving programming in which we asked "If you used the debugging strategy, did it help you find the problem?" The third source were two one-on-one interviews in which we asked each student, "What do you like about the strategies?", "What do you not like about the strategies?", and "What strategy do you prefer to use?" We performed these interviews just before the midterm and on the last day of class.

Measuring strategy use (RQ2) was challenging because strategies occur in the mind and are not directly observable. Therefore, we triangulated three sources of data. The first was the use of the Roboto strategy tracker tool. We used usage logging to determine whether students had stepped through any part of a strategy on each

day of instruction. The second source was help requests during class; we gathered data about whether students were using the strategy we taught, which we defined to students as "using the strategy tracker or reading the strategy to guide your work." The third source was self-report via an end of class survey asking students whether they used one of the strategies to solve a problem on that day, with or without the tracker. We collated all three data sources for each student, combining them into a single binary variable of whether the strategies had been used.

To measure the relationship between strategy use and productivity (RQ3), we measured productivity by analyzing students' final games for a set of game features (described later). To control for knowledge of JavaScript, we used the midterm scores assessing knowledge of variables, conditionals, Boolean logic, and Game Lab APIs. To measure strategy self-efficacy, our daily debrief survey asked students to express their agreement with the statements "I can follow the debugging strategy that [the instructor] taught to find and fix defects in my programs." and "I can follow the 'how to' strategy to find and adapt examples for my programs." on a 5-point scale of strongly disagree to agree. We asked this each day for each strategy that had been taught on that day or prior.

3 RESULTS

To contextualize our results, we begin by describing the classroom environment. Most students were engaged, but some were tired and slept, some occasionally used their smartphones, and some were distracted by sitting next to friends. The instructor and assistants wandered the lab, proactively offering help, responding to questions, and ensuring students were on task. Since most students spoke English as a second language, students moved at very different paces and the instructors and TAs regularly needed to give further explanation of the Code.org content. As suggested by prior work [11], the purely content-driven instruction in Code.org's curriculum was insufficient to produce robust knowledge of JavaScript's semantics, which led to many defects in students' programs.

3.1 RQ1: Barriers to using strategies

To answer RQ1, we performed several qualitative analyses. We followed Hammer and Berland's views on qualitative coding, not treating the results of our category generation as data itself, but as an organization of claims about data [9].

3.1.1 Strategy Interviews. The first analysis we performed was of the responses to our two interviews about the strategies, one prior to the midterm concerning the debugging strategy, and the second at the end of the course covering both strategies[1]. One author began by identifying categories of sentiments that students expressed about the strategies. This author then discussed the substance of the categories with the rest of the authors and resolved disagreements. The quotes we include paraphrase students' verbatim responses.

The first round of interviews about the debug strategy revealed that 46.7% of students preferred to guess the cause of defects and edit the program to verify their guess, 26.7% preferred asking for help, and 13.3% preferred reading and analyzing their code. Only 13.3% preferred the debug strategy. Out of the 13 distinct sentiments students expressed, only three were positive. Students claimed that they liked the step-by-step nature of the debugging strategy, feeling it helped them stay on track, one calling it a "formula for when you get stuck." Many students reported liking the debug strategy because it gave them an alternative to their ineffective editing strategy. One student said that it "forces us to actually look at our code instead of adding random stuff."

Although many students said that the debug strategy made it easier to solve their problems, the majority of their sentiments about it were negative. These included not being able to use the strategy independently, being unsure where to start, having difficulty identifying possible causes of defects, finding the strategy too time consuming, finding the strategy tracker interface confusing, disliking the tracker's design (having to log in, the website "looking boring"), and finding the strategy as too general and repetitive.

In the second round of interviews, conducted after 2 weeks of open-ended game development, 13.3% students preferred the debug strategy over guessing and editing (33.3%), asking for help (33.3%), and analyzing their code (20.0%). A chi-squared analysis on the frequency of the preferred strategies in the middle and end of the class showed no significant change in strategy preference (χ^2=0.64, p=.89). Students expressed many of the same sentiments about the debug strategy as before, but none of the students reported disliking the interface, being annoyed by having to log in, feeling unsure where to start, or feeling that the strategy was too general. Two new sentiments also emerged: students reported that the steps of the strategies were harder to remember without the tool, and that they disliked strategies rarely gave them solutions right away.

In responses about the reuse strategy, 60% of students claimed not having used it. Students preferred strategies including looking for resources without the tool (26.7%), asking for help (33.3%), and guessing implementation details (20.0%). Only 20.0% preferred the reuse strategy. Students reported liking that the reuse strategy helped them find solutions and that it prompted them to search for resources, whether online or from peers. They disliked, however, that it did not help them find solutions right away and found it harder to remember the steps without the tool. They also reported feeling like the reuse strategy was "pretty much cheating" because it used resources online, that they struggled to interpret the code they found online, that they often could not find resources, and that their lack of prior knowledge hindered their use of it: "kind of hard to know, because some of the things you're trying to learn you don't know yet."

3.1.2 Reactions to Strategy Modeling. To further understand barriers to using strategies, we analyzed notes about the 239 help requests for students' reactions to the help we provided when modeling strategy use. We conducted an inductive, qualitative coding of these notes, which involved generating and refining a set of categories from students' reactions, then classifying and resolving inconsistencies. Table 1 shows the range of reactions.

Students exhibited both positive and negative reactions toward the strategies. Positive reactions were characterized by compliance with, expressions of belief in, or an indication of learning as a result of the strategy. Negative reactions included impatience with the time strategy use required, frustration when the intended goal was not met, or inclination to revert back to guessing and editing.

[1]Our analysis omitted data from 2 students who missed one of the interviews.

Table 1: Categories of student reactions to strategy modeling

Category	Definition	Example notes about student reaction	Frequency
Reaction Not Recorded	Reaction to the help given was not recorded		53
Reaction Irrelevant to Strategy	Reaction was not regarding strategy usage		17
Indifference	No positive or negative feelings toward the strategy	"No reaction to mentioning the debug strategy"	15
Positive Sentiments			
Confidence in Strategy	Indication of a belief in the effectiveness of strategy use	"Seemed satisfied following the strategy and that it helped"	30
Compliance	Willing or enthusiastic use of the strategy	"Willingly participated in using the strategy tool"	19
Internalization	Ability to use strategy without the tool	"Was using the strategy in her head, but not online"	5
Negative Sentiments			
Reluctance	Indications of mild unwillingness to use the strategy	"Smiled and reluctantly agreed to using the strategy tool"	27
Dissatisfaction	Expressions negative emotions in response to the strategy	"He said 'I don't think [the strategy] is useful.'"	12
Resistance	Decision to revert to personal strategy	"Didn't use the strategy because he thought it wasn't helping"	16
Strategy Understanding			
Indicated Learning	Better understanding of the subject from strategy use	"He navigated to the causes and noticed missing parameters"	94
Indicated Difficulty	Confusion or missteps in the process of strategy use	"Didn't know possible causes"	38

These reactions reveal perceived barriers such as reluctance, dissatisfaction, and resistance. External factors, such as the strategy tracker's interface or strategy presentation, might have influenced sentiments, introducing other barriers.

Help request reactions also revealed expressions of struggle. These included difficulty completing the step of the debug strategy that required students to identify possible causes of failures and the step of the reuse strategy that required translating solutions into code. Conversely, students reported that they gained an ability to achieve a goal and indicated a greater understanding of the subject on which they received help by using a strategy.

3.1.3 Sentiments about Strategy Utility. We analyzed students' daily sentiments about the utility of the strategies based on the daily debrief surveys. We present these sentiments as percentages of the total student sentiments at the end of the course, added throughout each day the students were surveyed.

According to students' self-reported data over the whole class, 73% of students said they either agreed or strongly agreed to being able to understand the debug strategy, while 51% of students either agreed or strongly agreed to being able to understand the reuse strategy. Only 53% of the students said that the debug strategy was either helpful or somewhat helpful, while 46% of students said that the reuse strategy was either helpful or somewhat helpful. Only 1% of student responses claimed that both strategies were unhelpful and they were unable to follow said strategy. A large portion of the student responses claimed they never used a strategy: throughout the class, only 54% of students claimed they used the debug strategy, while 47% claimed they used the reuse strategy, at the end of the day. Fishers exact tests showed a significant association between students who reported using the each strategy and students who reported it helpful (both p<.00001).

3.2 RQ2: Scaffolding or tool?

To investigate whether the strategy tracker was scaffolding that eventually could be removed or a tool of consistent utility, we analyzed the use of the strategies over time and how they were used. Figure 4 and 5 show the debug and reuse strategy use over time respectively, including four measures. The first (blue) was the total number of students using the strategies, as indicated by a help request (yellow) in which someone was independently using the strategy, log data (green) showing the students using the strategy tracker, or self-reported use of the strategies (orange) in the daily surveys. These figures show that while use of the strategy tracker rapidly declined over several days, use of the strategies without the tracker continued throughout the course.

Figure 4: Debug strategy use

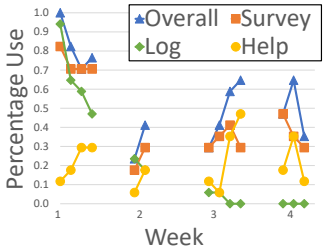
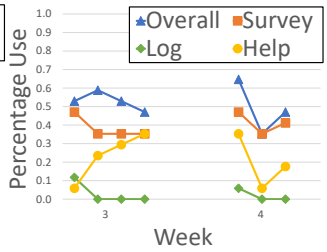


Figure 5: Reuse strategy use



3.3 RQ3: Is strategy use related to productivity?

Our last question concerned the relationship between strategy use and problem solving success. We defined productivity as successful implementation of game features. To identify these features, the teaching team analyzed all 13 final games as a group, generating a comprehensive list of game functionality. The final list of features and their respective frequencies included: keyboard input (13), collisions (13), score tracking (11), enemies (8), sound effects (8), scrolling background (6), lives (4), obstacles (2), gravity (3), animated sprites (8), falling objects (8), collectibles (7), levels (6), dynamic background music (4), title screen (6), game over screen (8), multiplayer (4), bouncing (3), instructions (1), mouse input (1), and restart functionality (1). Students' feature counts ranged from 7 to 12 with a median of 9.

To answer RQ3, we sought to model the relationship between strategy use and this productivity metric. We measured strategy use as the total number of days a student used a strategy via any of our three measures of strategy use and number of features implemented. We first checked for correlations between between game feature count and strategy use for each strategy. We found that using the debug strategy was significantly correlated with productivity (R^2 = 0.2627, p < 0.05), but using the reuse strategy was not (R^2 = 0.1414, p > 0.05).

Table 2: Ordinal logistic regressions

	Odds Ratio	SE β	Wald	Pr > χ^2
Debug Self Efficacy	0.70	0.75	0.23	0.63
Team	2.15	0.99	0.61	0.44
Debug Usage	1.56	0.36	1.54	0.22
Midterm Score	1.05	0.25	0.04	0.85
Help Request Count	0.98	0.07	0.08	0.77
Reuse Self Efficacy	0.55	0.68	0.76	0.38
Team	1.82	0.99	0.36	0.55
Reuse Usage	1.15	0.26	0.30	0.58
Midterm Score	1.01	0.26	0.00	0.96
Help Request Count	1.03	0.05	0.37	0.54

To account for other factors that may have mediated productivity, we next created ordinal logistic regressions for each strategy including more possible factors that influence the number of implemented features. The other factors we included were 1) prior knowledge of the Game Lab API based on midterm score, as brittle knowledge of the API would have limited productivity; 2) the most frequent response to the self-efficacy question for the strategy on the daily surveys after the midterm, as self-efficacy should have mediated successful use of the strategies; 3) whether or not they were on a team, as teams may have been more productive; and 4) the number of times they requested help from an assistant, as help seeking was likely to play a significant role in successfully building game features. Table 2 shows the resulting regressions. Neither model significantly explained the variation in productivity, suggesting that additional factors, or interactions between factors, were responsible for the number of game features students implemented.

4 DISCUSSION

Our case study revealed several trends. First, with respect to barriers (RQ1), all of our data sources suggest that while the adolescents in our class could see the merits of the strategies in the abstract, many did not see enough value to use them. Most chose to engage in rapid cycles of editing and testing, without deeply understanding their code, rather than the more systematic strategies we taught that required reasoning about program behavior. Student sentiments suggest that this was partly because of a perception that strategies slowed them down, but also because many of the skills the strategy required were something students' did not feel confident performing without help, such as identifying possible causes of defects or finding and reasoning about relevant code online. Our results on scaffolding (RQ2) suggest that having a tool that aided strategy execution may have scaffolded strategy learning over time, but that students who used the strategies appeared to do so by internalizing the them, rather than use the tracker. Our results on productivity (RQ3) suggest that while there was an association between using the debug strategy and how many game features students implemented, the relationship was not a direct one: there were likely many interacting factors that we did not model that determine whether students' were able to use explicit strategies to effectively guide their problem solving.

There are several possible interpretations of these results. One is that while explicit strategies may be more effective in principle, unless students believe they can perform them independently, they will be reluctant to use them, even when they have directly observed their benefits. A related interpretation is that providing effective, explicit strategies may only be effective for learners that have strong self-regulation skills, as prior work shows that learners with weaker regulation skills are often unaware of the need for better strategies [7, 10, 17, 20]. Another interpretation is an "attention economic" one [4]: in a classroom environment with substantial teaching support, asking for help is a more efficient strategy than trying to independently use a programming strategy a student has just learned; after all, both strategies had numerous failure modes that encouraged students to ask for help if they got stuck. Another interpretation is that the strategies were simply too sophisticated to learn while also learning basic programming concepts. Perhaps simpler strategies, scaffolded by instructor guidance, would be more likely to be adopted and more likely to impact behavior, even if they are less effective. For example, future work could explore a debugging strategy that simply prompts students to "find and understand the cause before editing," or a reuse strategy that prompts students to "ask an expert for an example and then work with them to adapt it to your needs." This level of strategic detail might be more appropriate for rank novices, and the level of complexity of problems they tend to face. Finally, perhaps most adolescents, who are known to have not yet fully developed executive functioning [2], do not yet have the self-regulation skills yet to delay gratification in a way that explicit programming strategies require.

Our study's limitations complicate these interpretations. Students might have been more positive in the interviews and surveys because of participant response bias. Our class was students' last of the day, and so many of their self-regulation skills may have been exhausted by a long day of math, writing, college prep, and other electives. The students we studied are also not representative of all adolescents learning to code.

Despite these limitations, our results have important implications for future research and teaching. First, our data suggest that there appear to be many interacting factors that influence adoption and use of explicit strategies in classrooms, such as students' willingness to delay gratification, the availability of help, the strength of their self-regulation skills, self-efficacy with the specific strategies taught, the likelihood of students' encountering problems that benefit from strategy use, and the alignment between the specific strategies taught and students' prior knowledge. This suggests that studying strategies in strictly controlled settings that omit these factors is not likely to be fruitful. Second, despite having a tool that carefully taught the strategies, an instructor that patiently explained the strategies, and an entire team of teaching assistants model the use of strategies with individual students, most students did not adopt the strategies. If we are to succeed in teaching adolescents programming strategies we know to be effective, future work must invent more effective, scalable ways to teach strategies. We hope our results are a solid foundation for this work.

ACKNOWLEDGMENTS

We thank our study participants for their time, the Upward Bound team for their support, and Dastyni Loksa for his feedback on the study design. This work was supported in part by the National Science Foundation under grants CCF-1703734 and CCF-1703304.

REFERENCES

[1] Roger Azevedo and Jennifer G Cromley. 2004. Does training on self-regulated learning facilitate students' learning with hypermedia? *Journal of Educational Psychology* 96, 3 (2004), 523.

[2] Roy F. Baumeister and Kathleen D. Vohs. 2003. Self-regulation and the executive function of the self. *Handbook of self and identity* 1 (2003), 197–217.

[3] Suresh K. Bhavnani, Frederick A. Peck, and Frederick Reif. 2008. Strategy-Based Instruction: Lessons Learned in Teaching the Effective and Efficient Use of Computer Applications. *ACM Transactions on Computer-Human Interaction* 15, 1, Article 2 (May 2008), 43 pages. https://doi.org/10.1145/1352782.1352784

[4] Alan F. Blackwell. 2002. First steps in programming: A rationale for attention investment models. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2.

[5] Ryan Chmiel and Michael C. Loui. 2004. Debugging: From Novice to Expert. In *ACM SIGCSE Technical Symposium on Computer Science Education*. 17–21. https://doi.org/10.1145/971300.971310

[6] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

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

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA.

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

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

[11] Ada S Kim and Andrew J Ko. 2017. A pedagogical analysis of online coding tutorials. In *ACM SIGCSE Technical Symposium on Computer Science Education*. 321–326.

[12] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *Comput. Surveys* 43, 3 (2011), 21.

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

[14] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Andrew J. Ko. 2018. Explicit Programming Strategies. *In review* (2018).

[15] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin* 36, 4 (June 2004), 119–150. https://doi.org/10.1145/1041624.1041673

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

[17] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem solving, and self-awareness: effects of explicit guidance. In *ACM SIGCHI Conference on Human Factors in Computing Systems*. 1449–1461.

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

[19] Devon H. O'Dell. 2017. The Debugging Mindset. *ACM Queue* 15, 1, Article 50 (Feb. 2017), 20 pages. https://doi.org/10.1145/3055301.3068754

[20] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.

[21] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[22] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2018. Assessing Bootstrap: Algebra Students on Scaffolded and Unscaffolded Word Problems. In *ACM Technical Symposium on Computer Science Education*. 8–13.

[23] Benjamin Xie, Greg L. Nelson, and Andrew J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *ACM Technical Symposium on Computer Science Education*. 344–349. https://doi.org/10.1145/3159450.3159527