A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension

Lasang Jimba Tamang
University of Memphis
Memphis, TN, USA
ljtamang@memphis.edu

Zeyad Alshaikh
University of Memphis
Memphis, TN, USA
zlshaikh@memphis.edu

Nisrine Ait Khayi
University of Memphis
Memphis, TN, USA
ntkhynyn@memphis.edu

Priti Oli
University of Memphis
Memphis, TN, USA
poli@memphis.edu

Vasile Rus
University of Memphis
Memphis, TN, USA
vrus@memphis.edu

ABSTRACT

We present in this paper the results of a randomized control trial experiment that compared the effectiveness of two instructional strategies that scaffold learners' code comprehension processes: eliciting Free Self-Explanation and a Socratic Method. Code comprehension, i.e., understanding source code, is a critical skill for both learners and professionals. Improving learners' code comprehension skills should result in improved learning which in turn should help with retention in intro-to-programming courses which are notorious for suffering from very high attrition rates due to the complexity of programming topics. To this end, the reported experiment is meant to explore the effectiveness of various strategies to elicit self-explanation as a way to improve comprehension and learning during complex code comprehension and learning activities in intro-to-programming courses. The experiment showed pre-/post-test learning gains of 30% (M = 0.30, SD = 0.47) for the Free Self-Explanation condition and learning gains of 59% (M = 0.59, SD = 0.39) for the Socratic method. Furthermore, we investigated the behavior of the two strategies as a function of students' prior knowledge which was measured using learners' pretest score. For the Free Self-Explanation condition, there was no significant difference in mean learning gains for low vs. high knowledge students. The magnitude of the difference in performance (mean difference = 0.02,95% CI: -0.34 to 0.39) was very small (eta squared = 0.006). Likewise, the Socratic method showed no significant difference in mean learning gains between low vs. high performing students. The magnitude of the performance difference (mean difference = -0.24,95% CI: -0.534 to 0.03) was large (eta squared = 0.10). These findings suggest that eliciting self-explanations can be used as an effective strategy and that guided self-explanations as in the Socratic method condition is more effective at inducing learning gains.

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432423

CCS CONCEPTS

• **Applied computing** → **Education**; *Interactive learning environments*; *Computer-assisted instruction*; *Computer-managed instruction.*

KEYWORDS

instructional strategies, free self-explanation, socratic method, learning gain, program comprehension, learning programming, intro-to-programming, computer science education

ACM Reference Format:
Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Priti Oli, and Vasile Rus. 2021. A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432423

1 INTRODUCTION

A key challenge in undergraduate Computer Science (CS) programs is high attrition rates. At a global level, a recent study [8] indicated that mean worldwide failure rate in these courses in 2019 was 33%. In the United States, Introductory CS courses (e.g., CS1 and CS2) often have attrition rates of 30-40% or even higher [7, 21, 51]. In contrary, local and global demand for skilled programmers have increased substantially and is expected to grow even more, e.g., the United States Bureau of Labor and Statistics predicts software developer job grows at the rate of 21% until 2028 [1]. Thus, it is imperative to develop and explore effective instructional strategies that will help students better understand the programming concepts and achieve higher learning gain which in turn will improve their chances of successful completion of CS1 and CS2 courses.

To this end, CS education researchers have spent considerable effort exploring effective instructional interventions that facilitate students for learning programming [22, 23, 36, 42, 45, 54]. A large number of automated tool papers are frequently published for instruction support. For example, tools that perform Automated Program Repair [32]. [52] found that while the graders seem to gain benefits from automatically generated repairs shown as hint, novice students do not seem to know how to effectively make use of it. [2] found that advantage of automatic feedback over manual feedback to resolve errors in code is primarily logistical and not conceptual;

the performance benefit seen during lab assignments disappeared during exams wherein feedback of any kind was withdrawn. Thus, tool that help novice programmer has to be developed (as mostly CS1 and CS2 students are novice programmers) and any strategy used in such tool should support long term mastery of concept or better mental model construction (rather than just immediate support for bug fixing). In order to do so, we need to understand the nature of programming and why teaching novices how to program remains a challenging task [26], which are discussed below.

The primary reason that large number of students fail in CS1 and CS2 courses is the inherent complexity of CS concepts itself [33]. Programming is considered a complex cognitive activity where a student has to simultaneously build and apply several higher order cognitive skills for solving a particular problem [42]. First encounters between a learner and a computer system are nothing short of a "shock" [18] as programming is a highly complex process involving a multitude of cognitive activities and mental representations related to problem understanding, programming methods, program design, program comprehension, change planning, debugging, and the programming environment. CS concepts are considered more complex than those of other fields traditionally considered challenging, such as mathematics. Thus, it is not surprising that many students in introductory programming courses feel overwhelmed.

The secondary hindrance in students' path towards mastery or learning of programming concept is difficulties with constructing accurate mental models [31]. Indeed, a major challenge in CS education is the difficulty novice programmers face with constructing accurate mental models during key learning activities, such as source code comprehension [29, 37, 39, 49]. This challenge is not surprising given that constructing mental representations is considered a higher-level skill of comprehension, typically engendering a high cognitive load[20, 25, 48, 55]. The importance of building accurate mental models during learning tasks has been well established for decades in domains like science [12, 16, 17, 34] as well as in CS education [29, 37, 39, 49].

In this work, with our goal of finding effective instructional strategies that help CS1 and CS2 students better learn programming concepts and build more accurate mental, we experiment with 1) eliciting free Self-Explanations and 2) a Socratic method that uses a sequence of questions that follow a guided line of reasoning to elicit guided self-explanations. We also investigate the effectiveness of these two strategies for students with two different levels of prior knowledge (low vs. high prior knowledge) as measured by a pretest. The following are our main research questions.

- ***RQ1***: Does eliciting free self-explanations help learners better comprehend instructional code examples? Is there a discrepancy between low vs high prior-knowledge students with respect to the impact of this strategy of eliciting self-explanations on student performance and learning?
- ***RQ2***: Does a Socratic method for eliciting self-explanations help learners better comprehend instructional code examples? Is there a discrepancy between low vs high prior-knowledge student with respect to the impact of this strategy on student performance and learning?
- ***RQ3***: Which of the two strategies are most effective: eliciting free Self-Explanations or the Socratic method for eliciting self-explanations?
- ***RQ4***: How do mental models of low prior knowledge students differ from those of the high prior knowledge group?

We hypothesize that both self-explanation elicitation strategies improve comprehension of code examples (resulting in more accurate mental models) and that both will lead to learning gains. This is based on self-explanation theories and prior evidence about the positive effect of self-explanation on learning and problem solving in science domains such as biology [11], physics [14], math [3], and programming [9, 50]. It should be noted that there could be many ways to prompt for self-explanations and understanding which types of prompting works best for whom is still an open question for code comprehension and learning. In other domains, several types of prompts have been studied already such as eliciting free explanation or justification-based self-explanation prompts [15] and meta-cognitive self-explanation prompts [11]. In our case, we study two different strategies for eliciting self-explanations: free Self-Explanations and guided Self-Explanations using a Socratic method that relies on a sequence of guiding questions. The free Self-Explanation strategies simply asks students to self-explain their reading and understanding of given code and then make predictions about the output of the code. The Socratic method has the same overall goal of eliciting self-explanations during reading and understanding of code examples. The Socratic method differs in that it draws more attention to key aspects of the code using a series of guiding questions. There is prior evidence that best teachers use a Socratic method [13].

The results of the randomized controlled trial experiment indicate that the learning gains of the free Self-Explanation strategy was 30% (M = 0.30, SD = 0.47) whereas for the guided Self-Explanation based on the Socratic method was 59% (M = 0.59, SD = 0.39). The Socratic method outperformed the free Self-Explanation strategy by 29 % - the difference was statistically significant ($p < 0.05$ level). For both the free Self-Explanations and the Socratic method, there was no significant difference in mean learning gains for low vs. high prior-knowledge students. Our key findings are: 1) Both eliciting free Self-Explanations and Socratic method is an effective intervention for learning programming and 2) both strategies help equally low and high prior-knowledge students.

While investigating the effectiveness of eliciting self-explanations in the area of CS Education has been explored before under certain experimental conditions and types of prompting and instructional activities such as problem solving, the main contributions and novel aspects of our work is the comparison of free Self-Explanation and of the Socratic method for code comprehension and learning. There has been no any prior work in comparing these two forms of self-explanations in the CS Education literature to the best of our knowledge.

2 RELATED WORKS

Self-explanation theories indicate that students who engage in self-explanations, i.e. explaining the target material to themselves, while learning are better learners, i.e., learn more deeply and show highest

learning gains. Self-explanation's effectiveness for learning is attributed to its constructive nature, e.g., it activates several cognitive processes such generating inferences to fill in missing information and integrating new information with prior knowledge, and its meaningfulness for the learner, i.e. self-explanations are self-directed and self-generated making the learning and target knowledge more personally meaningful, in contrast to explaining the target content to others [44]. The positive effect of self-explanation on learning has been demonstrated in different science domains such as biology [11], physics [15], math [3], and programming [9, 50].

A series of studies [9, 38, 40] found that self-explanations help learning Lisp programming concepts. They found that skill improvement had strong correlation with the amount of self-explanation generated. Two other studies with undergraduate students [41] and high school students [4] found that students who used a self-explanation strategy while studying worked out examples were more successful at a program construction task (Visual Basic) compared to those who did not apply the strategy. Effectiveness of the self-explanation in programming was also studied for SQL[53], JavaScript[27], HTML[28] and assembly language[24]. Study [9] showed that university students who underwent explicit training on strategies of self-explanation and self-regulation outperformed students in a control group (no explicit training) on problem solving performance.

In our case, we explore the role of two strategies for eliciting self-explanations for code comprehension and learning. In particular, our work is part of our larger efforts to explore the role of an advanced, online education technology to train learners on using self-explanations as a way to improve code comprehension processes and outcomes (accurate mental models, learning). More specifically, we explore two particular strategies for eliciting self-explanations: free or open-ended self-explanations versus guided explanations using a Socratic method [5, 6] that elicits self-explanations by using a series of guiding questions that draw attention to key aspects of a given code.

Indeed, there are different ways to elicit self-explanations which result in different types of self-explanations such as spontaneous self-explanations (no prompting), free or open-ended self-explanations (simple prompting to self-explain), guided (see the socratic method description in section 3.1.2), and scaffolded self-explanations (in this case students are encouraged to self-explain as much as possible by themselves and offered support in the form of hints when floundering). Other forms of self-explanations have been tried such as "complete given self-explanations (fill-in the blank self-explanations)" [28] and "select a self-explanation/menu-based self-explanations"[3, 19] which one may argue are not true self-explanations as the learner does not generate the explanation, i.e., the 'self' part of the 'self-explanation' is missing. Furthermore, self-explanation prompts can emphasize various aspects of self-explanations resulting in justification-based self-explanation prompts [15] or meta-cognitive self-explanation prompts [11]. Again, we explore here two types of self-explanations for code comprehension and programming concept learning: eliciting free or open-ended self-explanations and guided self-explanations using a Socratic method.

The Socratic method has been adopted for instruction in many domains, however, it has been rarely used for computer programming instruction. A study reported by [43] compared the effectiveness of a Socratic method versus didactic tutoring in a simulated problem solving environment for teaching basic electricity and electronics concepts. They found that participants in the Socratic condition learned more than students in the didactic condition. In the area of computer programming, there is one prior effort which studied the use of Socratic dialogue for teaching recursion and reported positive results [10]. Our study further investigates the effectiveness of the Socratic method for the purpose of developing an adaptive instructional system for code comprehension and learning. In our case, we targeted 6 concepts: operator precedence, nested $if-else$, for loops, $while$ loops, arrays, creating objects and using their methods. Targeting a broader set of computer programming concepts helps testing the generality of the Socratic methods across many concepts.

Importantly, to the best of our knowledge, there is no prior comparative study of free self-explanations versus the Socratic method and definitely not in the context of source code comprehension and learning programming concepts. The closest such study is [35] which examined the effectiveness of two learning strategies, self-explanation, and elaborative interrogation for the retention of scientific facts. The elaborate interrogation strategy is related to the Socratic method in the sense that both are based on using a line of questioning of the students eliciting elaborative answers from the students. They indicated that self-explanation participants significantly outperformed elaborate interrogation and repetition control participants on measures of cued recall and recognition.

3 COMPARATIVE STUDY

We conducted a randomized control trial experiment in which participants were assigned to three approximately equal experimental groups: a free Self-Explanation group, the Socratic method group, and a Prediction Only group. Then, participants were shown code examples and asked to either self-explain what the code does (free Self-Explanation) or answer a series of guiding questions related to key aspects of the code examples (Socratic method) or just predict the output of the code examples (Prediction Only). All participants took a pretest before being assigned to an experimental condition and posttest afterwards. The pre-/post-test scores were used to calculate learning gains as a measure for the effectiveness of the interventions. We used a web-based software system to run the experiment.

We conducted a one-way between group analysis of variance (ANOVA) to compare the mean learning gains of the experimental groups. Furthermore, we categorized participants into low and high-prior knowledge groups based on their mean pretest score for both the free Self-Explanation and the Socratic method groups separately. Then, we conducted an independent sample t-test to compare learning gains between the low and high prior knowledge groups. More details about group design, participants, materials used, experiment protocol and measures are given in the following sections.

3.1 Group Design

3.1.1 Free Self-Explanation. Participants in this condition were given one JAVA code example at a time. They were then asked to first self-explain in as much detail as they can their understanding of the code. An input box where they could type their self-explanation was available on the interface right below the given code example. The code examples were shown to the learners as a whole as opposed to one line at a time. Each code example focused primarily on a major programming concept such as loops. After they submitted their self-explanation, learners were asked to predict the output of the code. Once they submitted their prediction, they are shown the next code example, and the process repeats.

3.1.2 Socratic Method. Participants in this group experienced a Socratic method for reading and comprehending the code examples. They were shown one code example at a time and asked a series of questions that guided them in their process of examining the code and pay attention to key aspects such as the target topic or concept. These guiding questions for all 6 code examples used were developed by extensive discussions by our team of researchers. The ideal interaction with Socratic tutor looks like in figure 1.

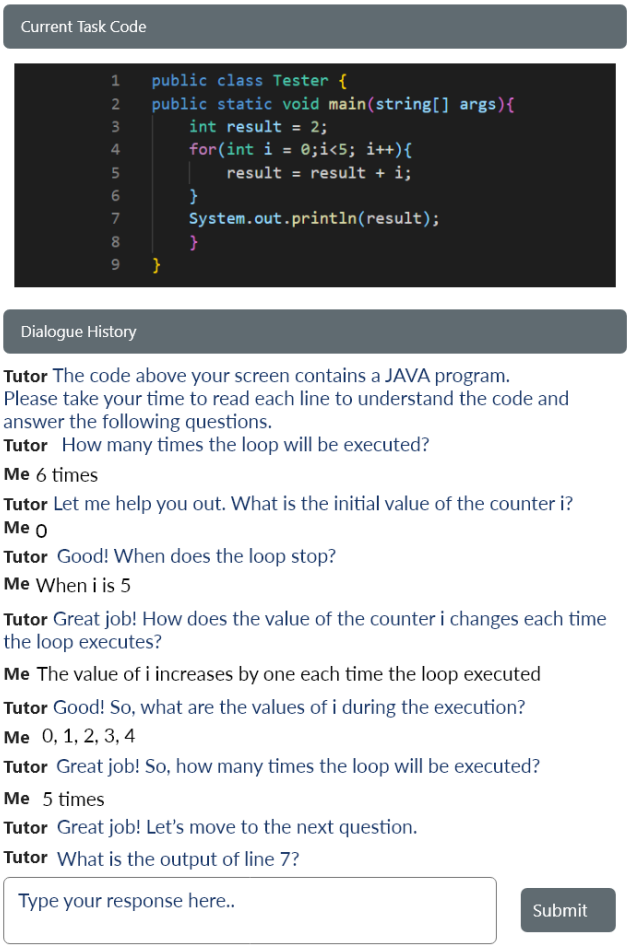

Figure 1: Interface for Socratic Method

3.1.3 Prediction Only. This is the control group. The participants in this group were shown code examples and asked to predict the output of the program. All participants in all condition were shown the correct prediction for each code example.

3.2 Participants

All 105 students of the third semester in an undergraduate program in computer science, 35 in each group, from an urban university in South East Asia took part in this experiment. All participants had undertaken courses on programming in prior semesters including structured programming C and they just had taken introductory Java course for two weeks at the time of this experiment. The participants were recruited with the help of the instructor and were paid for their participation. The instructor briefed student about the goal of the experiment including the fact that it is a good opportunity to solidify their programming skills. They were also told that the participation in the experiment is purely voluntary.

3.3 Materials

Participants in each of the experimental groups were shown same set of 6 source code examples. We call these 6 code examples the **main task**. Before this main task, participants were given a pretest that consisted of 6 code examples matching in terms of content, i.e., target concepts, the code examples in the main task. Furthermore, participants took a post-test consisting of 6 code examples matching in content that examples in the main task and pre-test. For the pre-test and post-test, learners were supposed to just provide the predicted output. As already noted, the pre-test and post-test were not identical, but they were equivalent in term of concepts tested and difficulty level. The main programming concepts covered by the experiment were: operator precedence, nested $if-else$, for loops, *while* loops, arrays, creating objects and using their methods. Each of these concepts were present in the code examples used in the pre-test, post-test, and the main task.

3.4 Protocol

The experiment was conducted in a computer lab in the presence of the instructor but not researchers of this study. Participants were told to ask questions, if required, only about the experimental procedure and system usage issues. The participants were first debriefed about the purpose and nature of the experiment and given a consent form. Upon their agreement, they were given a background survey followed by the pre-test. Then, they worked on the main task. Finally, they took the post-test. Participants could see all pretest and post-test questions at once on a single screen. They were shown the main tasks one at a time and they could proceed to the next task only after they submitted the answer for the current task. All participants' responses and interactions were automatically logged for post-hoc analysis.

3.5 Measures

To score the student performance on the main task, we used the predicted output for each code example. Each question was scored 1 if the final predicted output was correct and 0 for incorrect answers. This means the maximum score was 6 for pre-test, the main task, and the post-test, respectively. For each participant, we also

calculated the learning gains score using the following procedure [30].

- If posttest score > pretest score, learning gain = (posttest-pretest)/(6-pretest).
- If posttest score < pretest score, learning gain = (posttest-pretest)/pretest.
- If posttest score = pretest score = 6, discard the data.
- If posttest score= pretest score ≠ 6, learning gain = 0.

When posttest = pretest = 6, the learning gain = 0 - such scores are discarded, i.e., participant data with perfect scores of 6 in both pre-test and post-test is discarded. In our study, we ended with valid learning gains for 88 participants (28, 29 and 31 students in the Prediction only, Self-Explanation, and Socratic condition respectively). A total of 17 participants data points was discarded because they either achieved perfect scores in both pre-test and post-test (i.e. pretest = 6 and post-test=6) or they did not complete all parts of the experiment.

3.6 Results

The three randomly sampled experimental groups are equivalent in terms of pretest score (i.e. prior knowledge): they have same mean (M) pretest score of 3.4 with standard deviation (SD) of 1.95, 2.04 and 1.76 for prediction only, self-explanation and socratic method, respectively. Likewise, in the same order, the mean of posttest score is 3.57 (SD=1.53), 4.17 (SD=1.73) and 4.77 (SD=1.33) whereas the mean of task score is 3.29 (SD=2.26), 4.07 (SD=1.96) and 4.84 (SD=1.19). It is to be noted that any parametric techniques mentioned in this section 3.6 and section 4.1 met all the underlying assumptions, such as, normal distribution, random sampling, independence of observations, and homogeneity.

The result of ANOVA shows that there is a statistically significant difference ($p < 0.05$ level) in learning gains for the three groups (Self-explanation, Socratic method, and Prediction only): F(2,85) = 13.5, p=0.001. The difference in mean learning gain score is large, as suggested by [46]. The effect size, calculated using eta squared, was 0.24. Post-hoc comparison using Tukey's test indicated that mean score of the Prediction only (M = 0.047, SD = 0.31) was significantly different from the score of the Self-Explanation (M = 0.30, SD = 0.47). We also found that the Self-Explanation (M = 0.30, SD = 0.47) and the Socratic group (M = 0.59, SD = 0.39) were significantly different. There was also a significant difference between the Prediction only (M = 0.047, SD = 0.31) and the Socratic group (M = 0.59, SD = 0.39). *These results clearly indicate that both self-explanation (with average learning gain of 30%) and the Socratic method (with average learning gain of 59%) are effective at helping students learn computer programming concepts in JAVA. The Socratic method outperforms the free Self-Explanation by 29%.*

The results of the independent-sample t-test to compare the mean learning gain score for low and high prior-knowledge groups for Self-Explanation and Socratic Method are shown in Table 1 and Table 2, respectively. Note that any participants with pretest score <= mean pretest score for the groups falls into low prior-knowledge and the rest are in the high prior-knowledge group. The mean pretest score is 3.38 and 3.39 for Self-Explanation and Socratic method groups, respectively. Table 1 shows that there is no significant difference in mean learning gain score for the participants in the Self-Explanations group and the magnitude of the difference in mean learning gains (mean difference = .02,95% CI:-.34 to .39) is very small (eta squared = .006). Similarly, from Table 2, we can see that there is also no significance difference in mean learning gains for the participants in the Socratic method group and that the magnitude of the difference in mean learning gains (mean difference = -.24,95% CI:-.534 to .03) is large (eta squared = .10). *Thus, we found no significant evidence of discrepancy between low versus high prior knowledge participants for the two strategies (free Self-Explanation and Socratic method), respectively. That is, the two methods work equally well for students of all prior knowledge levels.*

Table 1: Independent Sample t-test result for learning gain between Low and High prior knowledge group in Self-Explanations

Group	N	Mean	SD	t-val	Sig.
Low Prior Knowledge	13	0.31	0.44	0.67	0.88
High Prior Knowledge	16	0.28	0.52		

Table 2: Independent Sample t-test result for learning gain between High and Low prior knowledge group in Socratic Method

Group	N	Mean	SD	t-val	Sig.
Low Prior Knowledge	14	0.45	0.31	0.67	0.87
High Prior Knowledge	17	0.70	0.43		

4 MENTAL MODEL ANALYSIS

To understand the mental model that students constructed during comprehending the code examples, we performed an in-depth analysis based on a qualitative inspection of the self-explanations which helps to reveal learner's comprehension at a finer grain level. This analysis was conducted only for the free Self-Explanations students as the Socratic method intervention led to shorter, less revealing responses due to the guiding questions which asked for specific responses of key aspect of the code. We randomly selected 5 low and 5 high prior-knowledge students from Self-Explanation group. Then, one of the authors of this experiment analyzed student's self-explanations along a number of factors that accounted for the quality of the self-explanations and mental models. The following factors were used for this qualitative analysis. These factors were based on self-explanation and code comprehension theories, e.g., the distinction between the program model, the domain model, and the situation model [37, 47].

Prior reference: Do they make any references to prior knowledge? Quality self-explanations entail integration of new information and prior knowledge.

Inferences: Do they make any inferences such as bridging inferences? For instance, when students explain one part of line of code, do they refer to prior lines or code or prior elements the code (bridging inferences)?

Monitor: Do they monitor and reflect on their understanding? Self-monitoring of one's comprehension is also a key aspect of quality self-explanations.

Control flow: Can students correctly identify the control flow of the program? Do they identify the order that function calls, instructions and statement execute or evaluates when program runs?

Data flow: Can students correctly identify the data flow and the state of all data at each moment of the code execution, i.e., how data structures are created, updated, or transformed?

Program model: Do students show an understanding of the major structural components of the code, i.e., the program model?

Domain model: Do students show an understanding of the domain model?

Integrated model: Do students show an understanding of the integrated model of the code by referring to links between the domain model and program model?

Mental model: This factor combines all the above into a holistic score: prior reference + inferences + monitor + control flow + data flow + program model + domain model + integrated model

The first three factors above are measured in terms of average (across all 6 main tasks) number of times learners make references to prior knowledge, number of inferences, and how many times they self-monitor their understanding, respectively. The rest of the factors (except mental model) were measured using 0-4 likert scale (0 - Very Poor, 1 - Below Average, 2 - Average, 3 - Above Average, 4 - Excellent) for each task and the final score used is the average score across all 6 main tasks.

Descriptive statistics and the results of an independent sample t-test were obtained to compare mean score for each of these factors between low and high prior-knowledge groups to understand any qualitative differences for the mental models those groups constructed.

4.1 Results

The results show that no student made references to prior knowledge or self-monitored their understanding. Only 5 students out of the 10 made altogether 19 inferences for all 6 tasks. Out of this, 18 inferences were made by 4 students in the high prior knowledge group and only 1 other student made 1 inference in low prior knowledge group. *This suggests that high prior-knowledge students make more inferences whereas low prior knowledge students do not.* Thus, training students to make more inferences could be possibly applied to make them more competitive.

Albeit using a small sample, we found a significant difference in mental model score for low prior knowledge group (M=4.99, SD = 1.05) and high prior knowledge group (M=8.67, SD = 3.57; t(8) =-2.233, p = 0.05, two tailed). The magnitude of the difference in the mean (mean difference =-3.88, 95 % CI: -7.7 to -0.03) was very large (eta squared = 0.38). Similarly, there is significant difference in control flow score between the low prior knowledge group (M=1.30, SD = 0.38) and the high prior knowledge group (M=2.37, SD = 0.99; t(8) =-2.237, p = 0.05, two tailed). The magnitude of the difference in the mean score (mean difference =-1.07, 95 % CI: -2.16 to -0.03) was very large (eta squared = 0.38). There was no significant difference in mean score for inferences, data flow, program model, domain model, or integrated model. *These results indicate high prior knowledge students are better than low prior knowledge group in using control flow to build better mental model.* Hence, low prior knowledge group could be trained more on control flow aspects of code reading and understanding to help them build better mental model which in turn help them to be more accurate.

5 CONCLUSION

We presented in this paper the results of a randomized control trial experiment that compared the effectiveness of two instructional strategies that scaffold learners' code comprehension processes: eliciting free Self-Explanation and a Socratic Method. The results showed pre-/post-test learning gains of 30% (M = 0.30, SD = 0.47) for the free Self-Explanation condition and learning gains of 59%(M = 0.59, SD = 0.39) for the Socratic method. For both self-explanations and Socratic method, there was no significant difference in mean learning gains for low vs. high prior-knowledge students i.e. no evidence of discrepancy by these interventions to students based on their prior knowledge.

We also analyzed students' comprehension using an in-depth analysis of their self-explanations and comprehension by assessing the quality of the self-explanations and the resulting mental models. These findings of this analysis suggest that students who make inferences and emphasize control flow are better comprehenders and learn more.

The Socratic method uses a sequence of guided questions emphasizing key aspects of target code which could be the reason for its better performance compared to the free Self-Explanation method which does not provide any specific hints. While the latter seem to be more revealing in terms of learners' comprehension processes and the resulting mental models, offering more support in the form of hints or guiding questions as in the Socratic method proves to be more beneficial to learning.

One of the limitations of our study is the coverage of CS topics in CS1 and CS2 courses. While this experiment focusing on 6 programming concepts was a good start to investigate and compare the effectiveness of free Self-Explanations and of the Socratic method, running a semester long experiment covering all intro-to-programming topics would be more conclusive. Furthermore, our in-depth analysis of the mental model students constructed needs to be extended to all students in the free Self-Explanation group as opposed to just a 5+5 sub-sample. It is our plan for future work to run a semester long study covering all topics in an intro-to-programming course and to extend our analysis of mental models to all students in the free Self-Explanations condition.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant number 1822816. All findings and opinions expressed or implied are solely the authors'.

REFERENCES

[1] [n.d.]. Software Developers : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. (Accessed on 08/19/2020).

[2] Umair Z Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd*

International Conference on Software Engineering: Software Engineering Education and Training. 139–150.

[3] Vincent AWMM Aleven and Kenneth R Koedinger. 2002. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive science* 26, 2 (2002), 147–179.

[4] Riyadh Alhassan. 2017. The Effect of Employing Self-Explanation Strategy with Worked Examples on Acquiring Computer Programing Skills. *Journal of Education and Practice* 8, 6 (2017), 186–196.

[5] Zeyad Alshaikh, Lasagn Tamang, and Vasile Rus. 2020. A Socratic Tutor for Source Code Comprehension. In *International Conference on Artificial Intelligence in Education*. Springer, 15–19.

[6] Zeyad Alshaikh, Lasang Jimba Tamang, and Vasile Rus. 2020. Experiments with a Socratic Intelligent Tutoring System for Source Code Understanding. In *The Thirty-Third International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*.

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

[8] Jens Bennedsen and Michael E Caspersen. 2019. Failure rates in introductory programming: 12 years later. *ACM Inroads* 10, 2 (2019), 30–36.

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

[10] Kuo-En Chang, Pin-Chieh Lin, Yao-Ting Sung, and Sei-Wang Chen. 2000. Socratic-dialectic learning system of recursion programming. *Journal of educational computing research* 23, 2 (2000), 133–150.

[11] Michelene TH Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian LaVancher. 1994. Eliciting self-explanations improves understanding. *Cognitive science* 18, 3 (1994), 439–477.

[12] Michelene TH Chi, Paul J Feltovich, and Robert Glaser. 1981. Categorization and representation of physics problems by experts and novices. *Cognitive science* 5, 2 (1981), 121–152.

[13] A Collins and A Stevens. 1982. Goals and methods for inquiry teachers. *Advances in instructional psychology* 2 (1982), 65–119.

[14] Cristina Conati and Kurt VanLehn. 2000. Further results from the evaluation of an intelligent computer tutor to coach self-explanation. In *International Conference on Intelligent Tutoring Systems*. Springer, 304–313.

[15] Cristina Conati and Kurt Vanlehn. 2000. Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. (2000).

[16] Ton de Jong and Monica GM Ferguson-Hessler. 1991. Knowledge of problem situations in physics: A comparison of good and poor novice problem solvers. *Learning and Instruction* 1, 4 (1991), 289–302.

[17] Andrea A DiSessa. 1993. Toward an epistemology of physics. *Cognition and instruction* 10, 2-3 (1993), 105–225.

[18] BENEDICT du Boulay. 2013. Some difficulties of learning to program. *Studying the Novice Programmer* (2013), 283.

[19] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2019. Evaluation of Parsons Problems with Menu-Based Self-Explanation Prompts in a Mobile Python Tutor. *International Journal of Artificial Intelligence in Education* 29, 4 (2019), 507–535.

[20] AC Graesser and DS McNamara. 2011. Computational analyses of multilevel discourse comprehension. Topics in Cognitive Science, 3 (2), 371–398.

[21] Mark Guzdial and Elliot Soloway. 2002. Teaching the Nintendo generation to program. *Commun. ACM* 45, 4 (2002), 17–21.

[22] Roya Hosseini, Kamil Akhuseyinoglu, Peter Brusilovsky, Lauri Malmi, Kerttu Pollari-Malmi, Christian Schunn, and Teemu Sirkiä. 2020. Improving engagement in program construction examples for learning Python programming. *International Journal of Artificial Intelligence in Education* 30, 2 (2020), 299–336.

[23] Roya Hosseini, Kamil Akhuseyinoglu, Andrew Petersen, Christian D Schunn, and Peter Brusilovsky. 2018. PCEX: interactive program construction examples for learning programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. 1–9.

[24] Yen-Chu Hung. 2012. Combining Self-Explaining With Computer Architecture Diagrams to Enhance the Learning of Assembly Language Programming. *IEEE Transactions on Education* 55, 4 (2012), 546–551.

[25] Walter Kintsch and CBEMAFRS Walter Kintsch. 1998. *Comprehension: A paradigm for cognition*. Cambridge university press.

[26] Yana Kortsarts, Kamil Akhuseyinoglu, Jordan Barria-Pineda, and Peter Brusilovsky. 2020. Integrating personalized online practice into an introductory programming course. *Journal of Computing Sciences in Colleges* 35, 8 (2020), 264–266.

[27] Kyungbin Kwon and David H Jonassen. 2011. The influence of reflective self-explanations on problem-solving performance. *Journal of Educational Computing Research* 44, 3 (2011), 247–263.

[28] Kyungbin Kwon, Christiana D Kumalasari, and Jane L Howland. 2011. Self-Explanation Prompts on Problem-Solving Performance in an Interactive Learning Environment. *Journal of Interactive Online Learning* 10, 2 (2011).

[29] Lauren E Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research*. 71–78.

[30] Jeffrey D Marx and Karen Cummings. 2007. Normalized change. *American Journal of Physics* 75, 1 (2007), 87–91.

[31] Iain Milne and Glenn Rowe. 2002. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies* 7, 1 (2002), 55–66.

[32] Martin Monperrus. 2020. The living review on automated program repair. (2020).

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

[34] Mitchell J Nathan, Walter Kintsch, and Emilie Young. 1992. A theory of algebra-word-problem comprehension and its implications for the design of learning environments. *Cognition and instruction* 9, 4 (1992), 329–389.

[35] Tenaha O'Reilly, Sonya Symons, and Heather MacLatchy-Gaudet. 1998. A comparison of self-explanation and elaborative interrogation. *Contemporary Educational Psychology* 23, 4 (1998), 434–445.

[36] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*. 204–223.

[37] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*. Ablex Publishing Corp., 100–113.

[38] Peter Pirolli and Margaret Recker. 1994. Learning strategies and transfer in the domain of programming. *Cognition and instruction* 12, 3 (1994), 235–275.

[39] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. 171–175.

[40] Margaret M Recker and Peter Pirolli. 1990. A Model of Self-Explanation Strategies of Instructional Text and Examples in the Acquisition of Programming Skills. (1990).

[41] Elizabeth Susan Rezel. 2003. The effect of training subjects in self-explanation strategies on problem solving success in computer programming. (2003).

[42] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

[43] Carolyn Penstein Rosé, Johanna D Moore, Kurt VanLehn, and David Allbritton. 2001. A comparative evaluation of socratic versus didactic tutoring. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 23.

[44] Marguerite Roy and Michelene TH Chi. 2005. The self-explanation principle in multimedia learning. *The Cambridge handbook of multimedia learning* (2005), 271–286.

[45] Vasile Rus, Peter Brusilovsky, Scott Fleming, Lasang Tamang, Kamil Akhuseyinoglu, Jordan Barria-Pineda, Nisrine Ait-Khayi, and Zeyad Alshaikh. 2019. An Intelligent Tutoring System for Source Code Comprehension. In *The 20th International Conference on Artificial Intelligence in Education, June 25-29, Chicago, IL, USA*.

[46] Shlomo S Sawilowsky. 2009. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods* 8, 2 (2009), 26.

[47] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. 2010. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*. 65–86.

[48] Catherine Snow. 2002. *Reading for understanding: Toward an R&D program in reading comprehension*. Rand Corporation.

[49] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.

[50] Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait-Khayi, and Vasile Rus. 2020. The Effects of Open Self-Explanation Prompting During Source Code Comprehension. In *The Thirty-Third International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*.

[51] Cameron Wilson, Leigh Ann Sudol, Chris Stephenson, and Mark Stehlik. 2010. Running on empty: The failure to teach K-12 computer science in the digital age. *Association for Computing Machinery* 26 (2010).

[52] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.

[53] Mashiho Yuasa. 1994. The effects of active learning exercises on the acquisition of SQL query writing procedures. (1994).

[54] Xihui Zhang, Chi Zhang, Thomas F Stafford, and Ping Zhang. 2019. Teaching introductory programming to IS students: The impact of teaching approaches on learning performance. *Journal of Information Systems Education* 24, 2 (2019), 6.

[55] Rolf A Zwaan and Gabriel A Radvansky. 1998. Situation models in language comprehension and memory. *Psychological bulletin* 123, 2 (1998), 162.