Exploring The Effectiveness of Reading vs. Tutoring For Enhancing Code Comprehension For Novices

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

Rabin Banjade
University of Memphis
Memphis, TN
rbnjade1@memphis.edu

Arun Balajiee Lekshmi Narayanan
University of Pittsburgh
Pittsburgh, PA
arl122@pitt.edu

Peter Brusilovsky
University of Pittsburgh
Pittsburgh, PA
peterb@pitt.edu

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

ABSTRACT

This paper presents a comparison of two instructional strategies meant to help learners better comprehend code and learn programming concepts: reading code examples annotated with expert explanation (worked-out examples) *versus* scaffolded self-explanation of code examples using an automated system (Intelligent Tutoring System). A randomized controlled trial study was conducted with 90 university students who were assigned to either the control group (reading worked-out examples, a *passive* strategy) or the experimental group where participants were asked to self-explain and received help, if needed, in the form of questions from the tutoring system(scaffolded self-explanation, an *interactive* strategy).

We found that students with low prior knowledge in the experimental condition had significantly higher learning gains than students with high prior knowledge. However, in the control condition, this distinction in learning outcomes based on prior knowledge was not observed. We also analyzed the effect of self-efficacy on learning gains and the nature of self-explanation. Low self-efficacy students learn almost twice as much in the interactive condition versus the passive condition although the difference was not significant probably because of low sample size. We also found that high self-efficacy students tend to provide more relational explanations whereas low self-efficacy students provide more multi-structural or line-by-line explanations.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Computing methodologies** → Natural language processing; • **Applied computing** → *Interactive learning environments*; *Computer-assisted instruction*;

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3636007

KEYWORDS

Intelligent Tutoring System, Self-explanation, Self-efficacy, Code reading, Code-comprehension

ACM Reference Format:
Priti Oli, Rabin Banjade, Arun Balajiee Lekshmi Narayanan, Peter Brusilovsky, and Vasile Rus. 2024. Exploring The Effectiveness of Reading vs. Tutoring For, Enhancing Code Comprehension For Novices. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605098.3636007

1 INTRODUCTION

The number of undergraduates who pursue a computer science major has increased dramatically [11]. Furthermore, there is an increasing demand for programming skills for non-CS majors. As a result, introductory CS courses, commonly known as CS1, are in high demand. However, various studies [8, 9, 12] reveal that these courses are challenging and have high attrition rates. Although the failure rates mentioned in these studies are questioned in some works [11], there is room for pedagogical improvement to meet the demand and standards for these courses. Our work presented here is a step in this direction.

Teaching and learning computer programming presents unique challenges and complexity compared to other subjects [28]. Fowler et al. [19] suggest that the difficulty of learning to program may have been worsened by an overemphasis on *code writing* at the expense of *code reading* proficiency. By prioritizing the former over the latter, novice programmers may have missed crucial opportunities to develop a deeper understanding of programming concepts and techniques. Lopez et al. [26] found that student performance on code reading and tracing questions accounts for 46% of the variance in their performance on code writing questions [26]. The study also found a correlation between the ability of a student to explain a piece of code using plain English and their ability to write similar code. Whalley et al. [43] argued that novice programmers must possess the ability to comprehend a given piece of code and its underlying knowledge and strategies before being able to write it. This requires understanding the code at a higher level of abstraction, such as its relational purpose, rather than simply focusing on its individual lines of code [43]. Indeed, to prepare students for writing code, activities like code tracing and code reading should be used

because they allow beginners to develop a conceptual understanding of basic programming skills with a lower cognitive load than writing code itself. For these reasons, our work focuses on pedagogical strategies to help students in CS1 courses better comprehend code and ultimately learn and master computer programming.

Another challenge in CS education, particularly in CS1 courses, is the need to scale instruction given that the number of CS and non-CS students has been increasing substantially. This could be addressed with the use of advanced learning technologies such as intelligent tutoring systems (ITSs) that can be deployed at scale and can provide effective tailored one-on-one instruction to every student. It is well documented that one-on-one instruction, i.e., *tutoring*, is among the most effective instructional methods [41]. When coupled with scalable, computer-based technologies, it results in scalable, effective instructional platforms.

The study presented in this paper investigates the effectiveness of scaffolded self-explanation as a tutoring strategy for code comprehension and learning by comparing it to the traditional self-guided reading strategy while accounting for other factors, such as student prior knowledge and self-efficacy. We experiment with a novel teaching and learning environment for CS education in the form of an Intelligent Tutoring System called *DeepCodeTutor* designed to help learners develop code comprehension skills and master programming concepts via scaffolded self-explanations. The development of DeepCodeTutor is based on self-explanation theories [15], the Socratic method of instruction guiding students' code comprehension and learning processes using a series of hints in the form of guiding questions which are triggered following constructivist learning theories according to which students construct their own knowledge and get help in the form of hints only when needed, and the ICAP framework [16], which postulates that *Interactive* instruction and learning is more effective than *Constructive* instruction and learning, which in turn is more effective than *Active* and *Passive* instruction and learning. Our work presented here is part of our larger goal of understanding the relationship between factors such as self-efficacy and prior knowledge, students' major (CS vs. non-CS), instructional strategies such as scaffolded self-explanations, and outcomes such as comprehension, learning, and retention.

The remainder of the paper is organized as follows. We start with reviewing the most relevant *related work*. In the subsequent section, we present the main *instructional material* and targeted computer science concepts, such as loops and arrays. The description of the *experimental system* follows, and then the *experimental design* and the results of the study are presented. We end the paper with conclusions and a discussion of future work.

2 RELATED WORK

In this section, we briefly highlight prior work on ITSs for computer science education, scaffolded self-explanation as a learning strategy, and comparative studies on the effectiveness of this approach compared to reading worked-out examples. We also discuss prior studies on the role of self-efficacy in learning. Bandura [6] defined self-efficacy as "people's judgments of their ability to organize and execute the courses of action required to attain designated types of performance. The relationship between self-efficacy and performance is straightforward. Students with high self-efficacy are more confident in their ability to learn and succeed, positively affecting their motivation and engagement and, ultimately, their overall performance [6]. Such students may need less scaffolding for their code comprehension and learning. The progress rate and overall performance are mediated by other factors, such as prior knowledge and instructional strategy. For instance, students with higher prior knowledge will more likely perform well early on, e.g., early successful steps on whatever instructional tasks they are working on, and make steady progress towards their learning goal. On the other hand, students with less prior knowledge will need more support early on. Adaptive educational technologies such as ITSs can monitor students' performance at each step while they engage in instructional activities and offer the support needed, at the right moment and in the right dosage, by each individual learner.

Indeed, a number of intelligent tutoring systems (ITS) have been developed to teach computer programming effectively. In 1986, Anderson and Skwarecki [4] developed the Lisp tutor, an intelligent tutoring system to provide tutoring to novice Lisp programmers where learners' input were continuously observed in order to provide scaffolding in case if the learner struggles or makes an error. In 2001, Graesser et al. [21] proposed AutoTutor, introducing a conversation agent to assist students in introductory computer literacy courses. AutoTutor is based on constructivist learning theories [1, 42] that assume that learning is more effective and deeper when learners actively generate explanations, justifications, and functional procedures, rather than being passive recipients of information to read. Alshaikh et al. [3] introduced a Socratic Tutor for the comprehension of source code for novices using a series of guiding questions derived from the abstract syntax trees of the statements in a code example. All students saw the same sequence of guiding questions regardless of their background. In our case, the guiding questions as hints are dynamically triggered by student performance, and thus different students will receive different guiding questions tailored to their needs. The questions prompt students to self-explain the code as a comprehension and learning strategy coupled with feedback from the tutor.

Self-explanation is explaining the learning material to oneself through speaking or writing [27]. Self-explanation is an effective strategy to help students learn and comprehend the target material in various domains [14, 15]. Studies with undergraduate [34] and high school students [2] have found that those who applied self-explanation techniques while studying worked-out examples performed better on programming tasks compared to those who did not use these strategies. Furthermore, the use of self-explanation strategies has shown an improvement in understanding programming concepts in introductory computer science courses [30, 38].

According to research conducted by Crippen et al. [17], pairing a worked example with a self-explanation prompt resulted in significant improvements in performance, problem-solving skills, and self-efficacy. A study [38] found that the Socratic method of guided self-explanation is more effective than free self-explanation in teaching novice code comprehension. Although self-explaining generally has positive effects on learning, it can be implemented more passively or interactively. In general, more interactive strategies are considered more effective according to the ICAP framework

[16]. In our scaffolded self-explanation strategy, students interact with an ITS and thus the effectiveness of their learning is improved.

The role of prior knowledge and its relationship to other factors such as the relative difficulty of the instructional tasks compared to students' knowledge level has been studied by VanLehn and colleagues [41] in the context of Newtonian Physics. They compared the effectiveness of different forms of instruction, including human tutoring (spoken and computer-mediated), natural language-based computer tutoring, and text-based control conditions, and found the tutorial dialogue to be more beneficial for novices studying content written for intermediates. However, when novices studied material written for novices or intermediates studied material written for intermediaries, the tutorial dialogue was not consistently more effective than text-based control conditions, which is confirmed by our study presented here. This suggests that the effectiveness of the tutorial dialogue may depend on the level of preparation of the student and the match between the student's preparation and the content of the instruction [18]. Although previous research suggests that increased student interaction, i.e., a more interactive learning/teaching strategy, such as one-on-one tutoring, leads to better learning outcomes [41], it is possible that other factors, such as motivation or verbal fluency, could influence both the student's interaction (e.g., during tutoring) and their learning gains. Our work will help to better understand the role of interactive learning and its relation and interaction with factors such as motivation, self-efficacy, and prior knowledge and their impact on various student outcomes. As noted above, our focus here is on evaluating the effectiveness of instructional strategies and examining the role of prior knowledge and self-efficacy in learning outcomes when employing different strategies.

Motivation in general and self-efficacy in particular is another important factor that impacts student performance and learning, which we considered in our work. According to a 2016 review [7], self-efficacy is the key factor in predicting the academic success of university students. Self-efficacy is a widely studied motivational factor in education, including computer science (CS) education. CS education researchers have found a strong connection between self-efficacy and CS student outcomes and have worked to improve methods of measuring self-efficacy in CS settings [25].

According to Bandura [6], an individual's perceptions and beliefs about the efficacy of a course of action affect an individual's decision to pursue or continue a task, the level of effort put forth on the task, tenacity in the face of task challenges, and overall task performance. Bandura further suggests that individuals with high domain self-efficacy are more likely to decide to take on complex work, put up more effort to complete it, and continue when the activity becomes more difficult.

Self-efficacy also influences the types of strategies individuals use and the coping mechanisms they use to deal with challenges and obstacles. Cernusca and Price [13] used a path analysis model to demonstrate that self-efficacy, perceived engagement, and perceived difficulty are all important predictors of students' final performance in CS1. Ramalingam and Wiedenbeck [33] developed a 32-item instrument to measure self-efficacy in computer programming, which consists of four subscales: "Independence and Persistence," "Simple Programming Tasks," "Self-Regulation," and "Complex Programming Tasks". These factors represent an individual's belief in their ability to work independently, complete basic programming tasks, regulate their own learning, and tackle complex programming challenges. Ramalingam et al. [32] expanded their earlier research on self-efficacy in computer programming by investigating the role of previous experiences, self-efficacy, and mental models in shaping students' performance in an introductory programming course. Their study examined how these factors can influence an individual's success in learning to program. Their findings indicate that self-efficacy in programming is influenced by prior programming experience and increases throughout the course and that the student's mental model of programming also influences self-efficacy. Lewis et al. [22] conducted a study to understand the factors that influence the student's decision to major in computer science. The study found that students' self-assessment of their programming ability which is based on their prior experience, programming tasks efficiency, and course grades influences their persistence to a computer science major. In a recent study, Lishinski [24] found that self-efficacy beliefs can create reciprocal feedback loops with performance, impacting the success of computer science students in their courses.

This paper builds on this prior work and explores the effectiveness of reading as compared to scaffolded self-explanation through our proposed intelligent tutoring system, the role of prior knowledge and self-efficacy in students' learning of programming concepts, and its interaction with instructional strategies such as scaffolding self-explanations.

3 CONTENT

For the main tasks in our experiment, we used a selection of Java code examples from the *DeepCode* codeset [35]. The *DeepCode* codeset is a collection of Java code examples annotated with explanations by experts and can be used for various purposes, including code comprehension and learning tasks in introductory programming courses. The *DeepCode* codeset was chosen for its strong theoretical foundations, which are based on a number of theories, including code comprehension and self-explanation theories and the ICAP framework [16, 40]. It was also explicitly designed to develop Intelligent Tutoring Systems that scaffold students' code-completion processes. These two factors make it an ideal choice for the code comprehension tasks in our study. The code examples in the *DeepCode* codeset include most topics in introductory programming courses, such as operators, loops, arrays, methods, and classes.

Each of the code examples in the *DeepCode* codeset is broken down into major logical blocks, each of which implements a clear functionality or sub-goal of the overall goal of the code. The explanations that accompany the code examples are divided into two main types: logical step and logical step details. These correspond to the domain and program models, respectively, of major code comprehension theories [37]. The details of the logical steps also link the domain model with the program model, thus, corresponding to the integrated model in code comprehension theories [37]. The code examples were designed to minimize the domain knowledge needed for full understanding and center around relatable, world-knowledge tasks or contexts, such as determining if a given year is a leap year. There is another type of explanation called statement-level explanations, which focuses on the new concept introduced

in each example. The code examples are sequenced so that each introduces only one new concept and relies on previously mastered concepts to manage the cognitive load. However, we do not use this sequencing in our work presented here as our experiment was just one session. The sequencing of topics and examples is important for semester-long use.

Additionally, the *DeepCode* codeset includes scaffolding questions for logical and statement-level expert explanations, which human or computer-based instructors can use as hints to scaffold students' comprehension and learning. For instance, following socio-constructivist theories of learning, these hints in the form of questions are progressive, starting with vague hints and becoming more and more informative, eventually providing fill-in-the-blank-type hints. This socio-constructivist approach allows the student to construct their knowledge by themselves as much as possible and only provides help through hints in the right dosage only when the student is floundering. In our study, we extended the scaffolding questions used in the *DeepCode* examples to scaffold students' understanding in multiple dimensions of code comprehension as identified by [38]: *inferences, control flow, data flow, program model, domain model,* and *integrated model*. In sum, our tutoring system aims to assess and assist the student in developing a deep understanding of the target code examples and learning programming concepts.

4 SYSTEM DESCRIPTION

As noted above, we compared two approaches to study code examples. In the first approach, participants simply read Java code examples augmented by expert explanations (worked-out examples). In the second approach, the students worked with code examples using a conversational Intelligent Tutoring System (ITS) known as the DeepCodeTutor.

The DeepCodeTutor approach involved presenting students with individual Java code examples, each accompanied by a clear description of the program's goal. Students were asked to provide self-explanations for the given code. These explanations are supposed to describe the functionality of the code in terms of logical steps/subgoals (domain model of the code) and explain how those logical steps are implemented (program model of the code) and how these are interlinked (the integrated model). These initial self-explanations were subsequently evaluated using automated semantic similarity methods. These methods compared the student's explanations with expert-provided explanations, such as those found in the 'DeepCode' code examples. Semantic similarity assessment was performed at both the sentence and paragraph levels, employing various features for comparison [36]. These features included an alignment score, determined by the optimal sentence alignment using chunks and a branch-and-bound solution to the quadratic assignment problem, word embeddings, and unigram overlap with synonym checks, bigram overlap, and the BLEU score [31]. If the similarity score reached 0.5 or higher, the student's explanation was deemed correct, and they progressed to the next task. In cases where the similarity score ranged from 0.4 to 0.5, the student's explanation was considered partially correct, and additional scaffolding was provided to address the incorrect portions. The selection of the 0.5 threshold value was informed by previous research [23].

The goal of DeepCodeTutor is to help students comprehend and explain the logical step and logical step details of a given code. If a student provides a complete and correct explanation, they will receive positive feedback and a summary explanation of what the code does. If the student misses important aspects or has misconceptions, DeepCodeTutor will use scaffolding questions to guide their comprehension and learning. The number of hints provided may vary depending on the student's needs. For instance, initially, the hints are vague, and if the student still struggles, the hints will be more and more informative eventually ending with a bottom-out hint in the form of a fill-in-the-blank question. The system will simply present the explanation if the student cannot provide a correct answer to the bottom-out hint.

The user interface of DeepCodeTutor consists of the following components. The goal description for the Java code example is displayed in the top left corner of the app and is highlighted in red for immediate attention and easy visibility for students (Fig. 1, A). The interactive code editor (Fig. 1, B) displays the target code example that the student should study to understand and explain. The code example is divided into logical blocks/chunks separated by empty lines. When a question is asked about a specific block/line of code, as shown in the figure, the target block is highlighted in yellow. On the right side of the interface (Fig. 1, C), is a display box that shows the entire dialogue history, displaying the student's response in blue on the right and the tutor's response in green on the left. The student input box is in the bottom right corner of the interface(Fig. 1, D). While it is beyond the scope of this paper to present all the details of DeepCodeTutor's design, it is important to note that we conducted several pilot studies prior to our main study. The primary objective of these pilot studies was to assess the functionality of our intelligent tutoring system. These pilot studies involved small groups of graduate and undergraduate students (with sample sizes ranging from 5 to 10) and yielded positive results.

5 EXPERIMENTAL DESIGN

We conducted a randomized controlled trial experiment to explore the effectiveness of scaffolded self-explanations provided through the DeepCodeTutor. Students were randomly assigned to one of the two experimental conditions: the control group (worked-out examples) versus the experimental group (scaffolded self-explanation through DeepCodeTutor).

The experimental group worked with the DeepCodeTutor (Fig. 1) introduced earlier while the control group studied expert-annotated code examples from the *DeepCode* codeset. Figure 2 shows the prompt asking students to read the annotated code example in the control condition. In the control condition, participants could move on to the next task only after confirming that they had read the annotated code example and the accompanying explanations, which was also evaluated based on whether the student scrolled to the end of the annotated code example.

The two groups worked with the same Java code examples, that is, all participants were exposed to equivalent content. Participants in both groups were tested for their mastery of the target concepts. The pre-and post-tests consisted of predicting the output of five Java code examples, which were aligned with the five main experimental

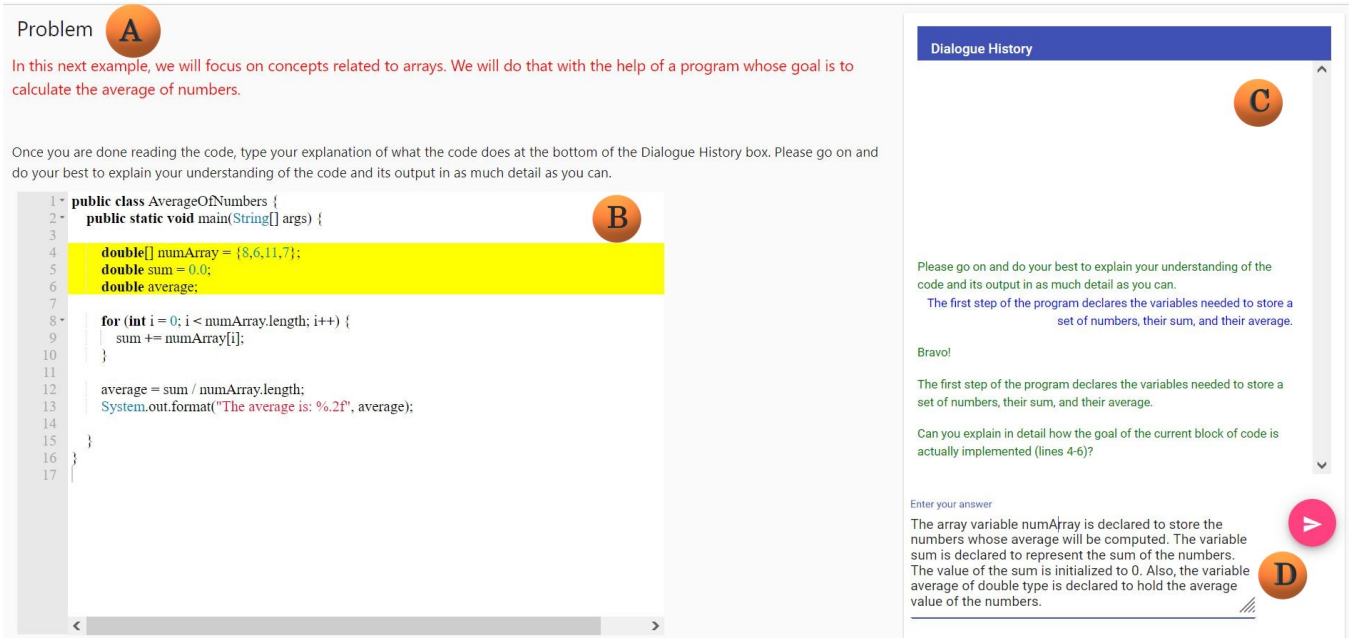

Figure 1: A screenshot of the DeepCodeTutor Interface: It includes (A) the goal description for each task, (B) an interactive Java code editor that shows the current Java code example, (C) a dialogue history of the interaction between the tutor and the learner, and (D) an input box for the learner to type their responses.

tasks. The performance of the students on these tests was used to calculate the main outcome variable: normalized learning gains.

5.1 Protocol

The study was carried out through Zoom[1] in groups according to the availability of the students. First, the students were briefed about the experiment, given the opportunity to ask questions, and then asked to sign a consent form if they agreed to proceed. Then they completed a background questionnaire about their main language of communication, programming experience, and current major. This was followed by a self-efficacy questionnaire and a pre-test, which assessed the students' prior knowledge of the programming concepts covered in the main experimental tasks: *variables and operator precedence, nested if-else statements, loops, arrays, and creating objects and using their methods.* The main tasks refer to the five code-reading tasks of the experiment.

Before starting the main task, the participants were provided with clear instructions on how to access and navigate the system used under their assigned conditions. After that, participants were placed in Zoom breakout rooms. They were requested to share their screens as they engaged in their assigned activities while closely monitored by a research coordinator (a graduate student). A graduate student was also available to answer questions during the experiment.

After completing the main tasks, the students took a post-test focused on the same concepts as covered in the pre-test and main tasks. Finally, the students completed an evaluation survey about their perception of the DeepCodeTutor. The system anonymously recorded all student input and tracked the time associated with each action.

[1]https://zoom.us

5.2 Participants

We recruited 90 students from an introductory Java programming class in an undergraduate Computer Science program at a large public university in the US. In the class, students learned Java programming through lectures, readings, and recitations. They also practiced Java knowledge through weekly homework assignments.

The participants' majors included Computer Science, Data Science, Computational Biology, Physics, Statistics, Engineering, Computational Social Science, Economics, and Statistics. Most of the participants were CS majors (n = 55). Fourteen participants identified themselves as non-native English speakers. Of the 90 participants, 89 completed the experiment, 47 in the control group, and 42 in the experimental group.

5.3 Instrumentation of Self-Efficacy

We assessed self-efficacy using a self-report survey using Askar and Davenport's computer programming self-efficacy scale [5]. This scale focuses on students' ability to learn and perform well in computer science courses, specifically their problem-solving confidence, debugging confidence, confidence in mastery, and confidence in receiving a good grade (i.e., success in the course). The student's response was recorded on a 5-point Likert scale: (1) Strongly disagree; (2) Disagree; (3) Neither agree nor disagree; (4) Agree; (5) Strongly agree for different questions. All questions were formulated with positive language. The questions used in the self-efficacy survey can be found in Table 4.

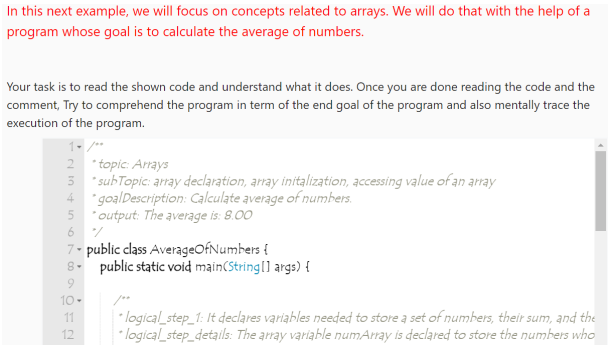

Figure 2: A screenshot displaying the prompt for reading the annotated code example in the control condition.

The Cronbach's Alpha for the ten items among all participants was found to be 0.85, which indicates that response values for each participant across all the items are consistent. To calculate the average self-efficacy score for each participant, the average of the responses to all questions was taken.

5.4 Performance Assessment

Normalized learning gain (NLG) [39] was used as the main performance metric to assess the effectiveness of the two different learning strategies, as it allows consistent analysis of diverse student populations with varying prior knowledge [29]. In our case, the NLG is calculated using the following formula (maximum possible post-test score = 5):

$$NLG = \begin{cases} \frac{posttest - pretest}{5 - pretest} & \text{if posttest > pretest} \\ \frac{posttest - pretest}{pretest} & \text{if posttest < pretest} \\ discard & \text{if posttest = pretest = 5 or 0} \\ 0 & \text{if posttest = pretest} \end{cases}$$

6 RESULTS

We organize this section around four major research questions our work has addressed.

RQ1: How Does the Effectiveness of Scaffolded Self-explanation using DeepCodeTutor Compare to Reading in Improving Code Comprehension Among Novice Learners?

We use normalized learning gain to measure the effectiveness of learning strategies. To compute the normalized learning gains for the two groups in our experiment, data from 21 participants in the control group and 11 participants in the experimental group were excluded because they had perfect pre-test and post-test scores. Similarly, the data of one participant was omitted from the analysis because they scored 0 on both the pre-test and the post-test. After eliminating these participants, the remaining participants in the study had an average pretest score of 3 (N=26, S.D = 1.23) in the control group and around 2.9 (N=30, S.D=1.41) in the experimental group. A t-test (t-val=-0.08, Sig. = 0.21) indicated that the two groups had similar levels of pre-test scores/prior knowledge. It is to be noted that any parametric techniques mentioned in this section (Section 6) met all the underlying assumptions, such as, normal distribution, random sampling, independence of observations, and homogeneity. Similarly, the t-test (t-val = -0.39, Sig.= 0.35) for the average self-efficacy for each group indicates that there is no significant difference between the average self-efficacy of the two groups. That is, the two groups were equivalent in terms of prior knowledge and self-efficacy.

In terms of normalized learning gains, we did not find significant differences between the control group (M=0.22, S.D=0.54) and the experimental group (M=0.26, S.D=0.40), although the students in the experimental group had a greater average learning gain (Table 1).

To better understand the effect size of the strategy implemented by DeepCodeTutor, we calculated Cohen's d, which was found to be a small effect size of 0.19 in favor of scaffolded self-explanation using DeepCodeTutor.

We also analyzed the distribution of learning gains in both groups using a heat map as shown in fig 3. Both groups exhibited notable learning gains specifically for the main task #2, which covered the nested if-else concept. For the main task #5, which covered the concept of classes and objects, the experimental group had an advantage over the control group.

Table 1: Independent sample t-test for the learning gains of the two experimental conditions.

Group	N	Mean	S.D	t-val	Sig.
Experimental Group	30	0.26	0.40	0.34	0.33
Control Group	26	0.22	0.54		

RQ2: Is the Effectiveness of Reading *Versus* Tutoring Influenced by Varying Levels of Prior Knowledge

We conducted a two-factor ANOVA to understand the effect of experimental condition and student prior knowledge. We assigned each student to a high or low prior knowledge group based on the median pretest score for each condition (Med = 3.5 for the experimental group and Med = 3 for the control group). We conducted a t-test on the pretest score between students with low prior knowledge (N = 15, M = 1.93, SD = 1.27) and students with high prior knowledge (N = 15, M = 4.06, SD = 0.25) under both conditions. The results indicate a significant difference in prior knowledge between the two groups ($t=6.32$, $p<0.05$), which validates our split using the median.

Table 2 shows the results of the two-factor ANOVA. It can be noted that the main effect of the "Group" factor was not statistically significant ($F(1, 52) = 0.17$, $p = 0.67$), indicating no significant differences in academic performance among the groups after implementing the tutoring strategy. However, the main effect of "Prior Knowledge" approached significance ($F(1, 52) = 2.93$, $p = 0.092$), suggesting that prior knowledge influences academic performance. In particular, the interaction effect between the "Group" and "Prior Knowledge" was significant ($F(1, 52) = 4.61$, $p = 0.03$), indicating that the impact of the tutoring strategy on learning varied based on the prior knowledge of the participants.

To further examine the impact of the scaffolded self-explanation delivered through the DeepCodeTutor on students with different

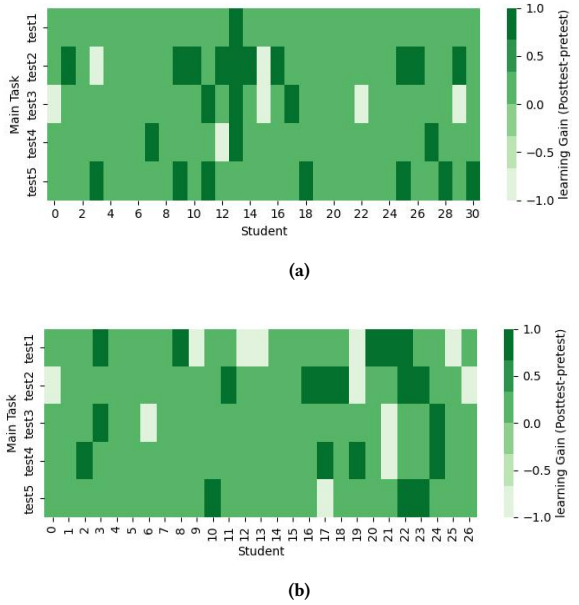

Figure 3: Heatmap of learning gain per question for students in (a) Experimental (b) Control group respectively

Table 2: Two-Factor ANOVA for the effects of Group and Prior Knowledge on the dependent variable - the normalized learning gains.

	sum_sq	df	F	PR(>F)
Group	0.03	1.0	0.17	0.67
prior_knowledge	0.62	1.0	2.93	0.09
group:prior_knowledge	0.51	1.0	4.61	0.036
Residual	11.05	52.0	-	-

levels of prior knowledge, we did a t-test. As we can see in Table 3, there is a significant difference in the normalized learning gain between low and high prior knowledge students in the experimental group, but no significant difference in the control group. This suggests that the scaffolded self-explanation may be particularly helpful for students with lower levels of prior knowledge which exhibited the largest mean learning gains. Figure 4 shows that students in the experimental condition with low-prior knowledge had a higher median and average learning gain than students in other conditions. The box plot also indicates that the spread of learning gain is greater among students with high prior knowledge in the control group, indicating more variation in learning gain within this group than in others.

RQ3: How Does A Student's Self-Efficacy Affect Their Learning When Using Different Learning Strategies: Reading *Versus* Scaffolded Self-explanations?

Table 3: Learning Gain Comparison for Low vs. High Prior Knowledge (Prior Knwldg) Student

Group	Prior Knw	N	Mean	S.D	t-val	Sig.
Experimental	Low	15	0.46	0.33	2.91	**0.003**
	High	15	0.07	0.39		
Control	Low	14	0.22	0.52	0.02	0.49
	High	12	0.22	0.58		

For RQ3, we first review the experimental results with an analysis of participants' self-efficacy. The average scores across all participants for all the self-efficacy questions are shown in Table 4. The control group's scores are higher for all questions except the last one. Participants reported the highest levels of self-efficacy in their ability to understand Java conditional expressions and trace well-defined iterative statements in Java. Conversely, participants reported the lowest levels of self-efficacy with respect to their mastery of Java programming concepts and their ability to achieve an excellent grade in a Java programming class.

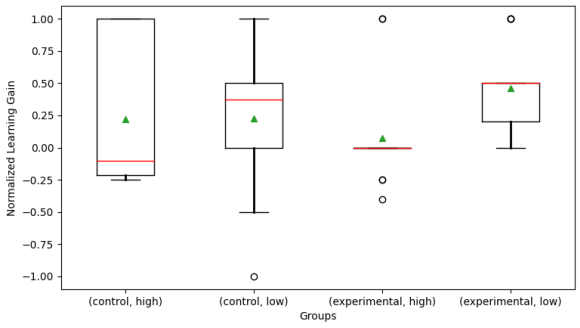

Figure 4: Box plot showing the normalized learning gain comparison among low-prior and high-prior knowledge students in the experimental and control conditions.

For the targeted concepts in our main experimental tasks (variables and operator precedence, nested if-else statements, loops, arrays, and creating objects and using their methods), self-efficacy scores are quite high (around 4.0 or above) indicating our sample is biased toward high self-efficacy students. The average self-efficacy for the experimental condition is lower than for the control condition but not significantly lower. We calculated the average self-efficacy for each group by taking the average of the self-efficacy across all items in each group. The average self-efficacy among all the participants was 3.96 (N=89, S.D = 0.48), 3.92 (N=42, S.D=0.48) in the experimental group and 4.01 (N=47, S.D=0.47) in the control group.

We also asked students to self-report their years of programming experience (not to be confused with professional programming experience) which can be viewed as a proxy for their prior knowledge. The correlation between self-efficacy and programming experience was 0.45 , and between self-efficacy and pretest scores was 0.47. These results suggest that students are relatively good

Table 4: Average and Standard Deviation (in parentheses) of self-efficacy per item.

Item	Overall Avg(S.D)	Control Avg(S.D)	Exp Avg(S.D)
I believe I will receive an excellent grade in Java programming class.	3.7 (0.88)	3.74 (0.93)	3.66 (0.83)
I have mastered the concepts taught in the Java programming class.	3.49 (0.8)	3.57 (0.81)	3.4 (0.78)
I can read Java programs and make changes to them according to some specified requirements.	4.0 (0.7)	4.10 (0.75)	3.88 (0.62)
I can write Java programs if given a specified set of requirements.	4.07 (0.7)	4.14 (0.68)	4.0 (0.72)
I can mentally trace through the execution of a long, complex Java program given to me.	3.47 (0.79)	3.55 (0.67)	3.38 (0.89)
I can understand Java's conditional expressions (e.g., if...else..).	4.56 (0.53)	4.57 (0.53)	4.54 (0.54)
I can mentally trace well-defined iterative statements in Java (e.g., for loop and while loop).	4.23 (0.68)	4.29 (0.61)	4.16 (0.75)
I can understand the concepts of objects and classes in Java, given a well-defined declaration of a Java class.	3.94 (0.67)	3.95 (0.68)	3.92 (0.66)
I understand how the array data structure works in Java and how to use it when coding.	4.21 (0.79)	4.29 (0.61)	4.11 (0.95)
I can debug(correct the errors) a long and complex program that I had written and make it work.	3.92 (0.76)	3.91 (0.79)	3.92 (0.73)

at self-reporting their prior knowledge, i.e., self-assessing their knowledge.

Table 5: Average and Standard Deviation(S.D) of self-efficacy grouped according to pretest score.

Pretest Score	N	Avg	S.D
5	36	4.13	0.4
4	23	4.02	0.31
3	16	3.95	0.59
2	3	3.46	0.12
1	7	3.45	0.49
0	4	3.37	3.29

The Table 6 presents the results of a two-factor ANOVA conducted to assess the effects of the experimental condition and self-efficacy on learning gains. The "Group" factor, representing the experimental condition, yielded a non-significant main effect ($F(1, 52) = 0.11$, $p = 0.73$), indicating that there is no significant difference in learning gain between the groups. Similarly, the "Self Efficacy" covariate showed a non-significant main effect ($F(1, 52) = 0.01$, $p = 0.89$), suggesting that self-efficacy did not have a significant impact on learning gains. The interaction between the experimental condition and self-efficacy was also non-significant ($F(1, 52) = 1.48$, $p = 0.22$), indicating that the combined effect of these variables on learning gains was not statistically significant. These findings suggest that neither the experimental condition nor self-efficacy exerted a significant influence on learning gain in this study.

	df	sum_sq	mean_sq	F	PR(>F)
Group	1.0	0.027	0.027	0.11	0.73
self efficacy	1.0	0.004	0.004	0.01	0.89
Group: self efficacy	1.0	0.33	0.33	1.48	0.22
Residual	52.0	11.84	0.22		

Table 6: Two-Factor ANOVA for the Effects of Group and self-efficacy on the Dependent Variable

To further investigate the effect of self-efficacy on learning gain and experimental condition, we divided students in each condition into two subgroups based on their self-efficacy scores: the low-self-efficacy and high-self-efficacy subgroups. The threshold for determining which subgroup a student belonged to was the median self-efficacy score, which was 3.9 for both groups (this median value was obtained after discarding some data as noted in the normalized learning gain analysis). This split resulted in a significant difference in both pretest scores and average self-efficacy between the subgroups (p-value < 0.05 in the control group and p-value < 0.05 in the experimental group), supporting our method to split the groups based on self-efficacy. Although the learning gain in high self-efficacy students is higher than that of the low self-efficacy students, the difference was not statistically significant as shown in Table 7.

Table 7: Learning gain comparison of low self-efficacy (S.E) versus high self-efficacy (S.E) students.

Group	N	Mean	S.D	t-val	Sig.
Low S.E	29	0.19	0.46	0.86	0.39
High S.E	27	0.30	0.48		

Table 8: Learning gain of high and low self-efficacy students in experimental versus control groups.

Group	Self-Efficacy	N	Mean	S.D	t-val	Sig.
Experimental	Low	15	0.28	0.36	0.14	0.88
	High	15	0.27	0.46		
Control	Low	13	0.13	0.55	0.85	0.39
	High	13	0.32	0.53		

Table 8 shows no significant difference in learning between students with low and high self-efficacy in the experimental group, suggesting that scaffolded self-explanation is equally effective for all levels of self-efficacy. Interestingly, high-efficacy students have similar mean learning gains in the two conditions. Importantly, Table 8 also shows that students with high self-efficacy had twice the learning gain compared to those with low self-efficacy when just

reading expert-annotated code examples, although this difference was not statistically significant. This may imply that the interactive strategy helps the low-efficacy students twice as much as the more passive strategy of reading experts' code explanations. The difference in learning, while large, is not significant and may be the result of our small sample size ($n = 13$ for control high self-efficacy and $n = 15$ for experimental low self-efficacy). This is something we will need to explore with a bigger sample size in the future.

RQ4: How Do Self Explanations Vary Among Students With Low and High Self-efficacy?

We further analyzed the nature of students' self-explanations while accounting for the self-efficacy level to see if there were any significant differences. The self-explanation was only collected for the participants in the experimental condition, as the control group only engaged in reading worked-out code examples. We analyzed students' first self-explanations, i.e., explanations generated freely without any scaffolding immediately after being prompted to read and self-explain their understanding of the code (before any scaffolding from the computer tutor). We calculated the volume of those first self-explanations by counting the number of content words (nouns, verbs, adjectives, and adverbs) after removing common stopwords (stopwords excluded common computer science terms such as *'for,' 'while,'* and *'if'* and so on). A t-test (t-stat=0.59,p=0.55) for the volume of self-explanation between low self-efficacy (M=22.5, S.D=17.21) and high self-efficacy (M=26.4, S.D=23.86) indicated no significant difference in the volume of explanations between the two groups.

Next, we analyzed the nature of self-explanations. According to Ramalingam and Wiedenbeck [32] there is a direct correlation between self-efficacy and mental models in computer programming. To this end, we analyzed the difference in explanations of low self-efficacy students to that of students with high self-efficacy. We used the SOLO taxonomy [10] to study differences in the nature of self-explanation among low- and high-self-efficacy students inspired by prior work which used the SOLO taxonomy to measure a student's understanding of a topic or concept [20].

We trained two graduate students on SOLO taxonomy. We asked them to rate students' free self-explanation (obtained on the first prompt) based on the five levels of understanding in the SOLO taxonomy. The inter-rater agreement between the two graduate students was found to be 0.87.

We observed that when self-explaining the code, high-self-efficacy students tend to provide more *relational explanations* (71%), which describe the relationship between different parts of the code. On the other hand, students with low self-efficacy tend to use a more *multi-structural or line-by-line approach* to self-explanation(68%). They focus on describing individual lines of code or actions taken rather than explaining how they relate to the overall logic of the program. For example, the following explanation is provided by a high self-efficacy student for a code to find the smallest divisor of a number: *" Two integer variables are declared, num and divisor. Num is assigned to the value 15, and the divisor starts with the smallest divisor larger than 1, 2. Then, the program runs through a while loop to find the smallest divisor of num."* This contrasts with the explanation of the same code by a student with low self-efficacy: *"First, assign the value 15 to the num variable. Then assign the value 2 to the divisor variable. After that, while the remainder of the num and divisor is not equal to 0, you keep adding one to the divisor"*. The difference in the explanations suggests a surface-level understanding of the code for low-self-efficacy students compared to high-self-efficacy students.

7 THREATS TO VALIDITY

One of the limitations of our study could be the relatively low difficulty of the main tasks compared to student mastery level. Indeed, most students (48%) had pre-test scores that were relatively high (4 or 5 out of a perfect score of 5), which means that our sample was biased towards high knowledge. Similarly, our sample was biased towards high-self-efficacy students. This may be the case because we recruited students at the end of the semester, i.e., students had an entire semester to master the programming concepts.

8 CONCLUSION AND FUTURE WORK

We examined the effectiveness of two instructional strategies (reading explained code examples vs. scaffolded self-explaining of code examples) and its impact on learning while students engage in code comprehension tasks.

Our findings suggest that students with low prior knowledge in the experimental group benefit significantly while using scaffolded self-explanation compared to students with high prior knowledge. In addition, the results of the experiment show a strong relationship between self-efficacy and prior knowledge, which means that students with low prior knowledge typically have low self-efficacy. In turn, they need more support in the form of hints and feedback for instructional purposes. The interactive strategy led to significantly better learning for low-prior-knowledge students, whereas learning gain of high prior knowledge students under the interactive condition was modest. This may be due to various reasons, including the mismatch between the task difficulty (too easy for high-knowledge students), which leads to some level of disengagement, including disengagement in the post-test.

For future work, we plan to investigate further the role of self-efficacy on learning and comprehension with a larger sample of students. Furthermore, since our sample was biased toward higher self-efficacy, we plan to change the timing of our experiment to the middle of the semester before students had too many opportunities to master the target concepts. Our results so far look promising, in particular, for low-knowledge students students who need more help, thus affirming the potential of advanced learning technologies that implement interactive instructional strategies to help students learn computer programming while engaging in code comprehension instructional tasks.

ACKNOWLEDGEMENT

This work has been supported by the following grants awarded to Dr. Vasile Rus: the Learner Data Institute (NSF award 1934745); CSEdPad (NSF award 1822816); iCODE (IES award R305A220385). The opinions, findings, and results are solely those of the authors and do not reflect those of NSF or IES.

REFERENCES

[1] 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.

[2] 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.

[3] Zeyad Alshaikh, Lasagn Tamang, and Vasile Rus. 2020. A Socratic Tutor for Source Code Comprehension. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part II 21*. Springer, 15–19.

[4] John R. Anderson and Edward Skwarecki. 1986. The automated tutoring of introductory computer programming. *Commun. ACM* 29, 9 (1986), 842–849.

[5] Petek Askar and David Davenport. 2009. An investigation of factors related to self-efficacy for Java Programming among engineering students. *Online Submission* 8, 1 (2009).

[6] Albert Bandura. 1986. Social foundations of thought and action. *Englewood Cliffs, NJ* 1986, 23-28 (1986).

[7] Kathryn Bartimote-Aufflick, Adam Bridgeman, Richard Walker, Manjula Sharma, and Lorraine Smith. 2016. The study, evaluation, and improvement of university student self-efficacy. *Studies in Higher Education* 41, 11 (2016), 1918–1942.

[8] 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.

[9] Jens Bennedsen and Michael E Caspersen. 2007. Failure rates in introductory programming. *AcM SIGcSE Bulletin* 39, 2 (2007), 32–36.

[10] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

[11] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (may 2017), 44–50. https://doi.org/10.1145/3084362

[12] Michael E Caspersen and Michael Kolling. 2009. STREAM: A first programming process. *ACM Transactions on Computing Education (TOCE)* 9, 1 (2009), 1–29.

[13] Dan Cernusca and Clayton E Price. 2013. Can undergraduates learn programming with a "Virtual Professor"? Findings from a pilot implementation of a blended instructional strategy. In *2013 ASEE Annual Conference & Exposition*. 23–268.

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

[15] 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.

[16] Michelene TH Chi and Ruth Wylie. 2014. The ICAP framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist* 49, 4 (2014), 219–243.

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

[18] Xianyong Fang. 2012. Application of the participatory method to the computer fundamentals course. In *Affective Computing and Intelligent Interaction*. Springer, 185–189.

[19] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding" Explain in Plain English" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1163–1169.

[20] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proceedings of the 17th ACM conference on international computing education research*. 69–80.

[21] Arthur C Graesser, Kurt VanLehn, Carolyn P Rosé, Pamela W Jordan, and Derek Harter. 2001. Intelligent tutoring systems with conversational dialogue. *AI magazine* 22, 4 (2001), 39–39.

[22] Colleen M Lewis, Ken Yasuhara, and Ruth E Anderson. 2011. Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In *Proceedings of the seventh international workshop on Computing education research*. 3–10.

[23] Mihai C Lintean and Vasile Rus. 2012. Measuring Semantic Similarity in Short Texts through Greedy Pairing and Word Semantics.. In *Flairs conference*. 244–249.

[24] Alex Lishinski. 2023. Self-efficacy Feedback Loops and Learning Experiences in CS1. (2023).

[25] Alex Lishinski, Sarah Narvaiz, and Joshua M Rosenberg. 2022. Self-efficacy, Interest, and Belongingness–URM Students' Momentary Experiences in CS1. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 44–60.

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

[27] Danielle S McNamara and Joseph P Magliano. 2009. Self-explanation and metacognition: The dynamics of reading. In *Handbook of metacognition in education*. Routledge, 60–81.

[28] 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.

[29] Jayson M Nissen, Robert M Talbot, Amreen Nasim Thompson, and Ben Van Dusen. 2018. Comparison of normalized gain and Cohen's d for analyzing gains on concept inventories. *Physical Review Physics Education Research* 14, 1 (2018), 010115.

[30] Priti Oli, Rabin Banjade, Arun Balajiee Lekshmi Narayanan, Jeevan Chapagain, Lasang Jimba Tamang, Peter Brusilovsky, and Vasile Rus. 2023. Improving Code Comprehension Through Scaffolded Self-explanations. In *International Conference on Artificial Intelligence in Education*. Springer, 478–483.

[31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[32] 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.

[33] Vennila Ramalingam and Susan Wiedenbeck. 1998. Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research* 19, 4 (1998), 367–381.

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

[35] Vasile Rus, Peter Brusilovsky, Lasang Jimba Tamang, Kamil Akhuseyinoglu, and Scott Fleming. 2022. DeepCode: An Annotated Set of Instructional Code Examples to Foster Deep Code Comprehension and Learning. In *International Conference on Intelligent Tutoring Systems*. Springer, 36–50.

[36] Vasile Rus, Sidney D'Mello, Xiangen Hu, and Arthur Graesser. 2013. Recent advances in conversational intelligent tutoring systems. *AI magazine* 34, 3 (2013), 42–54.

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

[38] 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*. 219–225.

[39] Ronald K Thornton, Dennis Kuhl, Karen Cummings, and Jeffrey Marx. 2009. Comparing the force and motion conceptual evaluation and the force concept inventory. *Physical review special topics-Physics education research* 5, 1 (2009), 010105.

[40] Kurt VanLehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.

[41] Kurt VanLehn, Arthur C Graesser, G Tanner Jackson, Pamela Jordan, Andrew Olney, and Carolyn P Rosé. 2007. When are tutorial dialogues more effective than reading? *Cognitive science* 31, 1 (2007), 3–62.

[42] Kurt VanLehn, Randolph M Jones, and Michelene TH Chi. 1992. A model of the self-explanation effect. *The journal of the learning sciences* 2, 1 (1992), 1–59.

[43] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, PK Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *8th Australasian Computing Education Conference (ACE2006), Australian Computer Science Communications*, Vol. 28. Australian Computer Society, 243–252.