# Improving Code Comprehension Through Scaffolded Self-explanations

Priti Oli[1(✉)], Rabin Banjade[1], Arun Balajiee Lekshmi Narayanan[2], Jeevan Chapagain[1], Lasang Jimba Tamang[1], Peter Brusilovsky[2], and Vasile Rus[1]

[1] University of Memphis, Memphis, TN 38152, USA
{poli,rbnjade1,jchpgain,ljtamang,vrus}@memphis.edu
[2] University of Pittsburgh, Pittsburgh, PA 15260, USA
{arl122,peterb}@pitt.edu

**Abstract.** Self-explanations could increase student's comprehension in complex domains; however, it works most efficiently with a human tutor who could provide corrections and scaffolding. In this paper, we present our attempt to scale up the use of self-explanations in learning programming by delegating assessment and scaffolding of explanations to an intelligent tutor. To assess our approach, we performed a randomized control trial experiment that measured the impact of automatic assessment and scaffolding of self-explanations on code comprehension and learning. The study results indicate that low-prior knowledge students in the experimental condition learn more compared to high-prior knowledge in the same condition but such difference is not observed in a similar grouping of students based on prior knowledge in the control condition.

**Keywords:** Program Comprehension · Scaffolding · Computer Science Education · Java Programming · Intelligent Tutoring System

## 1 Introduction

Computer programming is a critical skill in today's world. However, learning to program is challenging, as shown by high attrition rates (30–40%, or even higher) in introductory CS courses [1]. The premature focus on *writing* code rather than *reading* code may be part of the reason why learning to program has historically been challenging [3]. Code comprehension activities such as code tracing and code reading have been argued to be critical in the early stage of learning because they allow beginners to develop programming skills with a lower cognitive load than writing code itself [2].

In this paper, we present an Intelligent Tutoring System (ITS) *DeepCodeTutor* that helps students master code comprehension skills by providing automatic assessment and scaffolding for self-explanation of code examples. Self-explanation has been proven to be an effective strategy for learning computer programming concepts [6]; however, the presence of a human tutor is usually required to make


N. Wang et al. (Eds.): AIED 2023, CCIS 1831, pp. 478–483, 2023.
https://doi.org/10.1007/978-3-031-36336-8_74

this strategy work. In *DeepCodeTutor*, students' self-explanations are automatically assessed, appropriate feedback is provided (positive, neutral, negative), and, if required, a sequence of scaffolding hints follows. The hints are in the form of questions that prompt the student to think. We follow constructivist theories of learning with early hints in the sequence being vague and then more and more informative if students are still floundering.

In the following sections, we introduce the *DeepCodeTutor*, explain its approach to assessment and scaffolding of self-explanations, and present a randomized controlled study of its effectiveness in helping learners better understand code and master programming concepts.

## 2 DeepCodeTutor: Automatic Scaffolding of Code Explanations

DeepCodeTutor aims to help students comprehend and explain the logical step and logical step details of a given code example. The system engages students in a dialog-based approach, prompting students to explain the code of the worked-out program example and reacting to student explanations. If a student provides a correct and complete explanation, she will receive positive feedback and a summary explanation of the code. If the student's explanation is incomplete or incorrect, the system uses scaffolding questions to guide their comprehension and learning and correct misunderstandings. The number of hints provided varies depending on the student's individual needs, understanding, and articulation. The system will show the model explanation if the student fails to explain the concept correctly, even after scaffolding.

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. It 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 the student should read, comprehend, and articulate. 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, while the tutor's response is in green on the left. The student input box is at the bottom right corner of the interface (Fig. 1, D).

The tutor and the student discuss the code block by block. At the start of each task, the students are asked to explain the program in their own words. The student's initial explanation is then automatically assessed using automated semantic similarity methods, which compare the student's explanation to a benchmark, expert-provided explanation (e.g., the expert explanations in the DeepCode). The semantic similarity is computed by extending word-to-word semantic similarity measures to sentence and paragraph level [5]. The semantic similarity is calculated at the sentence and paragraph level by comparing a variety of features, including an alignment score based on the optimal alignment of

the sentences using chunks and a branch-and-bound solution to the quadratic assignment problem, word embeddings, unigram overlap with synonym check, bigram overlap and BLEU scores [5]. To be considered complete and correct, students must effectively convey all the key ideas in experts' self-explanations. If a student articulates a major misconception which DeepCodeTutor will detect, using a bank of major misconceptions, and correct it immediately.

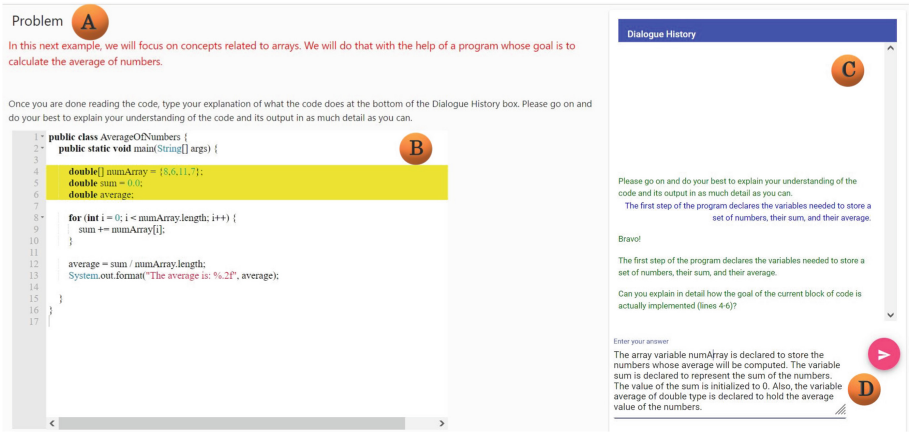


**Fig. 1.** A screenshot of the DeepCodeTutor Interface: It includes (A) The goal description for each task, (B) an interactive Java code editor that shows the current Java code example, and (C) a dialogue history of the interaction between the tutor and learners, and (D) an input box for the learner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

## 3 Experimental Design

To assess the value of DeepCodeTutor, we conducted a randomized controlled trial experiment in which participants were randomly assigned to two approximately equal groups: an experimental or treatment group that interacted with DeepCodeTutor (scaffolded self-explanations) and a control group in which participants were asked to read the expert-annotated code examples worked-out examples (no self-explanations elicited, no scaffolding offered). Both groups were presented with identical Java code examples, ensuring equal content exposure. The main outcome variable we focused on was the effectiveness of the two conditions in inducing learning gain, measured as an improvement in pre-to-post-test scores. The pre-and post-tests were identical and consisted of five short Java programs for which the student had to predict the output.

We recruited 90 students from an introductory Java Programming class in an undergraduate Computer Science program at a large public university in the United States. The participants were compensated with gift cards and extra credit for their participation. The study was conducted online by providing clear instructions about accessing and navigating the system. Students were asked to share their screens to ensure they followed the instructions correctly. A research

team member was available to assist with any questions during the experiment. Out of all the participants, 14 identified themselves as non-native English speakers. Out of the 90 participants, 89 completed the task, 47 in the control group and 42 in the experimental group.

The overall experiment protocol was as follows. First, students were informed about the experiment, offered a chance to ask questions, and asked to sign a consent form if they agreed to proceed. Then, they completed a background questionnaire about their primary language of communication, programming experience, and current major. This was followed by a self-efficacy questionnaire and a pretest, which assessed students' prior knowledge of the targeted programming concepts. After that, the subjects proceeded to the main task, where they interacted with five worked code examples doing either code-reading or scaffolded self-explanation, depending on the group. After completing the main task, the students took a posttest targeting the same concepts that were covered in the pretest and main task. Finally, the students completed an evaluation survey to provide their perceptions of TutorApp. The system logged all student inputs and tracked the time associated with each action without recording any identifiable information.

We used the Normalized Learning Gain (NLG) as the main performance metric to evaluate Learning Gain. It allows for consistent analysis of diverse student populations with varying prior knowledge [4]. For the calculation of NLG, if posttest score >pretest score, NLG = (posttest-pretest)/(5-pretest). If posttest <pretest, NLG = (posttest-pretest)/pretest. We discard all the cases where the student's score is perfect, i.e., 5 in both pretest and posttest. While the pretest and post-test scores are not perfect but equal, then the NLG = 0.

## 4 Results

For the normalized learning gain analysis, data from 21 participants in the control group and 11 participants in the experimental group were excluded due to perfect pretest and posttest scores, and also excluded one participant for scoring 0 on both tests.

Table 1 shows the average pretest, posttest, normalized learning gain, and learning gain (posttest-pretest) for the experiment and control group. The distribution of pretest scores indicated a bias in our sample towards high-prior knowledge students, with more students having a high pretest score. Also, the self-efficacy scores of the students in the study were found to be generally high, with a mean of 3.96 (S.D = 0.48) for all students, 3.92 (S.D = 0.48) for the experimental group, and 4.01 (S.D = 0.48) for the control group. This suggests that our data is skewed toward high-prior knowledge and high self-efficacy participants.

### 4.1 How Effective is Automated Scaffolded Self-explanation for Code Comprehension and Learning?

We first examined the differences in learning between the experimental and control group using a t-test, which showed no significant difference in the normal-

**Table 1.** Mean and standard deviation of pretest, posttest, and normalized learning gain (NLG) and learning gain (posttest-pretest).

| Group | n | Pretest | | Posttest | | NLG | | Learning Gain | |
|---|---|---|---|---|---|---|---|---|---|
| | | mean | SD | mean | SD | mean | SD | mean | SD |
| Experimental Group | 30 | 3.0 | 1.41 | 3.56 | 1.19 | 0.26 | 0.40 | 0.56 | 1.08 |
| Control Group | 26 | 3.19 | 1.23 | 3.53 | 1.20 | 0.22 | 0.54 | 0.34 | 1.14 |

ized learning gain ($t = 0.33$, $p = 0.36$) or the post-test score ($t = 0.08$, $p = 0.465$) between the two groups. To better understand the effect size of DeepCodeTutor, we calculated Cohen's d for the learning gain, which was found to be a small effect size of 0.19 in favor of scaffolded self-explanation. While we hypothesized that using DeepCodeTutor would lead to better learning, the study has not confirmed this hypothesis. The lack of significant differences could result from several factors, for e.g., our sample was biased toward high-prior knowledge and high self-efficacy students. Furthermore, we conducted the experiment at the end of the semester, which means students had many chances to master the concepts.

### 4.2 Does the Use DeepCodeTutor Result in Different Learning Outcomes for Students with High and Low Prior Knowledge?

To examine the impact of DeepCodeTutor on students with different levels of prior knowledge, we split students in each experimental condition into two subgroups based on their prior knowledge (Med = 3.5 for the experimental group and Med = 3 for the control group). We conducted a t-test on the pretest score between low-prior knowledge ($N = 15$, $M = 1.93$, $S.D = 1.27$) and high-prior knowledge students ($N = 15$, $M = 4.06$, $S.D = 0.25$) in the experimental condition. The results show 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.** Independent sample t-test for learning gain of low-prior and high-prior knowledge student in experimental and control group

| Group | Prior Knowledge | N | mean | SD | 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 | | |

As we can see in Table 2, there is a significant difference in the normalized learning gain between low and high prior knowledge students in the experimental group, whereas the average learning gain is the same for the control group. This suggests that the scaffolded self-explanation may be particularly helpful for students with lower levels of prior knowledge.

## 5 Conclusion and Future Work

In this paper, we presented a novel approach to engaging students in studying program examples and reported the results of its experimental evaluation. The key idea of our approach is to support student self-explanation of code fragments with automatic assessment and scaffolding. The results of the experiment show a statistically significant learning gain in low-prior-knowledge students in the experimental condition compared to high-prior-knowledge. In future work, we plan to investigate further the effectiveness of our technology.

**Acknowledgements.** This work has been supported by the following grants awarded to Dr. Vasile Rus: the Learner Data Institute (NSF award 1934745); CSEdPad: Investigating and Scaffolding Students' Mental Models during Computer Programming Tasks to Improve Learning, Engagement, and Retention (NSF award 1822816), and Department of Education, Institute for Education Sciences (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. Beaubouef, T., Mason, J.: Why the high attrition rate for computer science students: some thoughts and observations. ACM SIGCSE Bull. **37**(2), 103–106 (2005)
2. Lopez, M., Whalley, J., Robbins, P., Lister, R.: Relationships between reading, tracing and writing skills in introductory programming. In: Proceedings of the Fourth International Workshop on Computing Education Research. ACM (2008)
3. McCracken, M., et al.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, pp. 125–180 (2001)
4. Nissen, J.M., Talbot, R.M., Thompson, A.N., Van Dusen, B.: Comparison of normalized gain and Cohen's d for analyzing gains on concept inventories. Phys. Rev. Phys. Educ. Res. **14**(1), 010115 (2018)
5. Rus, V., D'Mello, S., Hu, X., Graesser, A.: Recent advances in conversational intelligent tutoring systems. AI Mag. **34**(3), 42–54 (2013)
6. Tamang, L.J., Alshaikh, Z., Khayi, N.A., Oli, P., Rus, V.: 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, pp. 219–225 (2021)