The Thirty-Third International
FLAIRS Conference (FLAIRS-33)

The Effects of Open Self-Explanation Prompting During Source Code Comprehension

Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Vasile Rus
Department of Computer Science, Institute of Intelligent System
University of Memphis, Memphis, TN, USA
{ljtamang, zlshaikh, ntkhynyn, vrus}@memphis.edu

Abstract

This paper reports the findings of an empirical study on the effects and nature of self explanation during source code comprehension learning activities in the context of learning computer programming language Java. Our study shows that self explanation helps learning and there is a strong positive correlation between the volume of self-explanation students produce and how much they learn. Furthermore, self-explanations as an instructional strategy has no discrepancy based on student's prior knowledge. We found that participants explain target code examples using a combination of language, code references, and mathematical expressions. This is not surprising given the nature of the target item, a computer program, but this indicates that automatically evaluating such self-explanations may require novel techniques compared to self-explanations of narrative or scientific texts.

Introduction

Computer Science (CS) education is critical in today's world where computing skills, such as computer programming, have become an integral part of many disciplines, including the fields of science, math, engineering, and technology. Although such skills are in high-demand, and the number of aspiring CS students is encouraging, a large gap between the supply of CS graduates and demand persists. For example, college CS programs suffer from high attrition rates (30-40%, or even higher) in introductory CS courses (e.g., CS1 and CS2) (Guzdial and Soloway 2002; Beaubouef and Mason 2005; Wilson 2010).

One reason for the high attrition rates in CS1 and CS2 courses is the inherent complexity of CS concepts and tasks (Du Boulay 1986; Morrison, Margulieux, and Guzdial 2015). 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. (Morrison, Margulieux, and Guzdial 2015) argue that using textual languages (to name and keep track of variables and handle related processes) while at the same time understanding and controlling an external agent (i.e., the computer) involves a level of complexity not seen in science, math, or engineering. Thus, it is not surprising that many students in introductory programming courses feel overwhelmed.

This work is part of a project whose aims are to develop effective and engaging instructional interventions to improve comprehension and learning in introductory Computer Science courses at college level, to reduce attrition rates and increase retention, and to ultimately produce more and better-trained graduates. The result will be a win-win-win situation for aspiring students, CS programs and their organizations, and the overall economy. Furthermore, the plan is to incorporate these effective and engaging intervention in advanced education technologies such as intelligent tutoring systems (ITSs), such as, (Rus et al. 2019).

One of the working hypothesis of our project is instructional strategies such as eliciting self-explanations will result in mental models that are more accurate, which in turn will positively impact comprehension, learning, self-efficacy, and retention (Chi et al. 1994; Best et al. 2005; Ramalingam, LaBelle, and Wiedenbeck 2004). The positive role of self-explanations is well documented for science learning but less so for computer programming learning. Therefore, our work fill this gap by answering the following broad research questions:

- ***Role of Self-Explanation***: Does self-explanation help in learning of computer programming? Is there any relationship between the volume of self-explanation generated by learners and their learning gains? Does self-explanation as teaching strategy has discrepancy in teaching low and high prior knowledge student?
- ***Nature of Self-Explanation***: How exactly do students explain source code? Are there any peculiarities and major differences compared to self-explanations of, say, narrative or scientific texts?

Understanding the role of self-explanation during source code comprehension could provide an effective instructional strategy for addressing the high-attrition rates in introductory computer programming courses. Additionally, by studying the nature of self-explanations during source code comprehension instructional activities will enable us to understand the challenges that need to be addressed in order to

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

develop methods to automatically assess student responses during such code comprehension activities. This in turn will enable the development of intelligent tutoring systems that could implement self-explanation elicitation strategies, automatically evaluate students' responses based on which tailored feedback and support could be provided to each learner. The preliminary results presented in this paper are the first step towards understanding the effectiveness and the nature of self-explanations students generate during source code comprehension activities and for building fully automated intelligent tutoring systems to help students in introductory to computer programming courses.

Background and Related Works

Self-explanations are learner-generated explanations of learning materials (McNamara and Magliano 2009; Crippen and Earl 2004; Van Merriënboer and Sluijsmans 2009; Roy and Chi 2005). Self-explanations usually include metacognitive statements, e.g., learners' own judgment of their level of understanding, and inferences based on the information from the learning materials and learners' own knowledge. According to Roy and Chi (Roy and Chi 2005), several cognitive mechanisms are involved: generating inferences to fill in missing information, integrating information within the learning materials, integrating new information with prior knowledge, and monitoring and repairing faulty knowledge. The mix of self-assessment and inference leads to improved understanding, more coherent mental models, and better learning compared to activities of learning where students, for instance, just read the instructional materials.

There are three major types of self-explanations: spoken or thinking out loudly, typed or written down, and silent reflection. The first two types are widely used. In the thinking out loudly type of self-explanations, the learner thinks aloud about the instructional materials presented to them (McNamara 2009). This type of self-explanation has been found to benefit proficient readers in general compared to less proficient readers (Muñoz et al. 2006). For written self-explanation, learners write down their thoughts while engaging with a particular learning material such as reading scientific texts to understand and learn target concepts or while trying to solve a problem(Muñoz et al. 2006). This form of self-explanation has been found to be more suitable for learners who have difficulty with demanding reading tasks such reading of scientific texts that requires a higher cognitive load (as opposed to less demanding reading tasks such as reading narrative texts). In our case the instructional material is computer code which we can make an argument it is more demanding than reading scientific texts. Indeed, Computer Science is considered even higher than science and math when it comes to cognitive load. For this reason that understanding source code requires a high cognitive load, we opted for the written-down type of self-explanations. Importantly, written or typed self-explanations were shown to enable readers to make more inferences, e.g., bridging inferences that links a target instructional material to prior knowledge, as opposed to text-bound processes such as paraphrases (Muñoz et al. 2006). In particular, when typing their explanations of science texts less skilled readers were more inclined to make bridging inference compared to speaking self-explanations. Typing seems to afford readers more time to reflect and access and express their thoughts. Our work here contributes to this line of research by exploring the role of typed explanations for a novel task: source code comprehension.

Despite the fact that self-explanation are found to improve the construction of mental models (Chi et al. 1994; Ramalingam, LaBelle, and Wiedenbeck 2004), very few studies have been conducted to investigate the effectiveness in doing so. A series of studies (Recker and Pirolli 1990; Pirolli and Recker 1994; Bielaczyc, Pirolli, and Brown 1995) found that self-explanations help learning Lisp programming concepts (their experimental population was undergraduate students). While they found that skill improvement had strong correlation with the amount of self-explanation generated, they also noted that the type of explanation also accounts for the improvement in performance: explanations of high performing students were much more structured into goal-based episodes compared to those of low performing students. This suggests that analyzing in more depth the nature self-explanations could help us better understand their impact on performance and how that impact is mediated by other factors such as students' prior knowledge. We investigated in the study reported here both the effectiveness of self-explanations and their nature.

Study with undergraduate(Rezel 2003) and high school(Alhassan 2017) students found that students who used self explanations while studying worked out examples were more successful at a program construction task (Visual Basic) compared to those who did not apply it. Further studies for JavaScript(Kwon and Jonassen 2011), HTML (Kwon, Kumalasari, and Howland 2011) and assembly language(Hung 2012) also found that self-explanations helped in learning programming. Our work contributes to this line of research on the role of self-explanation in learning JAVA programming. Furthermore, most notably all of the programming languages studied in the past are procedural language while we study JAVA which is Object Oriented Programming Language. To the best of our knowledge, there are no studies on the role of self explanations to learn Java.

Experimental Setup

Our goal was to study both the role of self-explanations during program comprehension tasks as well as the nature of self-explanations. For this reason, our experiment was so designed to collect self-explanations from all participants. On the other hand, to study effectiveness of self-explanations, it was necessary that some participants worked on code examples with self-explanation and others without it so that we can see the effect of the strategy on students' learning. To balance both needs, we randomly assigned participants to one of the following two experimental groups.

Self-Explanation First: Participants in this group interacted with an online system that showed them 4 code examples in Java. For each such code example, participants were prompted to predict its output and explain their thinking in writing. Then, they were shown another 4 code examples

for which they were simply asked to predict the output of the code examples (no explanation, just prediction).

Prediction First: Participants in this group followed the same procedure as the participants in the Self-Explanation First condition except the order of the prediction and self-explanation tasks was switched. That is, participants were asked to just predict the output of 4 code examples first. Then, they were shown 4 additional code examples for which they are asked to also explain the shown code examples.

It should be noted that we focus on open self-explanations as opposed to supported/scaffolded self-explanations. That is, we simply encourage students to self-explain while reading code examples without any further support, e.g., we do not assess their self-explanations and do not provide feedback and hints in case they are any misconceptions detected or the self-explanations are vague and incomplete. We opted for the open self-explanation because we wanted to explore the nature of students' freely generated explanations, i.e., when they are simply encouraged to self-explain without much guidance.

Materials

All participants first provided answers to a background questionnaire and then took a pretest (6 code examples for which they had to predict the output). After that, each participant was randomly assigned to one of the two experimental groups described above. At the end, all participants took a posttest (predict the output of 6 Java code examples). The pretest and postest were not identical but they were equivalent in term of concepts tested and difficulty level. The posttest was created from the pretest just by altering variables values or by minimal alternation of certain parts of the code such as the condition of $while$ or for loops. 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 pretest, post-test, and the experimental tasks.

Participants

A total of 26 college students (7 female and 19 male) from an urban university in Asia took part in the experiment. The participants were briefed about the goal of the experiment including the fact that they could solidify their programming skills by participating in the experiment. The participants were all volunteers and received a small compensation for their participation in the experiment. All participants were in the fourth semester of an undergraduate program in computer science and had undertaken same courses in their prior semesters. In addition, these participants have fair understanding of computer programming concepts as they took C and C++ courses in prior semesters. However, they have not taken JAVA course yet. Participants were randomly assigned to one of the two experimental groups using a group-balancing approach (half of the participants were assigned to one experimental condition and half to the other).

Procedure

The experiment was conducted in a computer lab in the presence of two experimenters. All the materials were shown using a web-based system via browser. They were only allowed to ask questions related to the experimental procedure and the use of the web-based system. The participants were first debriefed about the purpose and nature of the experiment, and given a consent form. Upon their agreement, they were shown the background survey followed by pretest. Then, they worked on the main part of the experiment that showed them prediction tasks and self-explanation tasks in the order corresponding to the experimental group in which they were randomly assigned. Finally, they worked on the posttest. Participants could see all pretest and posttest questions while working on the corresponding sections. However, they were shown the prediction or self-explanation experimental tasks one at a time; 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.

Assessment

Each question/task was scored 1 if the answer was correct; otherwise 0. This means the maximum score was 6 for the pretest, 4 for the self-explanation tasks, 4 for the prediction tasks, and 6 for the posttest. For each participant, we also calculated the learning gain score as suggested by (Marx and Cummings 2007). If posttest>pretest, gain = (posttest-pretest)/(6-pretest). If posttest <pretest, gain = (posttest-pretest)/pretest. If posttest=pretest= 0 or 6, drop the cases. If posttest=pretest, gain = 0.

Results

Out of 26 participants, we dropped data from 3 participants because they had a perfect score in both pretest and posttest. Overall, participants had an average of 1.7 (SD/stddev = 1.5) years of programming experience, 0.07 (SD = 0.31) years professional programming experience, a score of 3.3/6 (SD = 1.9) in pretest, and 3.87/6 (SD = 1.63) in postest. We also computed their scores for the main experimental tasks. The average score was 2.2 (SD = 1.53) for the 4 prediction tasks, 1.61 (SD = 1.37) and a learning gain score of 0.31 (SD = 0.42), calculated as suggested by (Marx and Cummings 2007). More detailed analysis are presented below. All the t-test applied below in our study met the standard assumptions of t-test (continuous scale for dependent variable, random sampling, independence of observations, normal distribution and homogeneity of variance).

Role of Self-Explanation

The goal of our study was to test the hypothesis that self-explanation helps both program comprehension and learning. Furthermore, we investigated whether the more participants self-explain the more they learn or not.

Does Self-Explanation enhance learning of core computer programming concepts? To answer this question, we first check equivalency between Prediction First and Self Explanation First group in terms of their prior knowledge

Group	N	Mean	SD	t-val	Sig.
Self Explanation	13	2.9	1.91	-1.31	0.20
Prediction	10	3.9	1.92		

Table 1: Independent sample t-test result of pretest scores between Self-Explanation First and Prediction First groups.

Group	N	Mean	SD	t-val	Sig.
Self Explanation	13	2.54	1.67	2.17	0.042
Prediction	10	1.30	1.06		

Table 2: Independent sample t-test result for the prediction score for self-explanation first and prediction first group.

i.e. pretest score, by applying independent sample t-test. The result in table 1 shows that groups are equivalent. Now, to answer our question, we performed an independent-sample t-test between Self-Explanation First and Prediction First group to compare their prediction scores. The results in table 2 shows that there is a statistically significant difference in prediction score for self-explanation first(M=2.54, SD=1.67) and prediction first group(M=1.3, SD=1.06; t(20)=1.62, p=0.042). The magnitude of the difference in the means (mean difference = 1.24, 95% confidence interval: 0.05 to 2.43) is large (Cohen's d = 0.9) as suggested by (Sawilowsky 2009).

Participants in Self-Explanation First group performed (2.54-1.3)/4*100=31% better, as measured by prediction score, than participants in the Prediction First group. An analysis of covariance (ANCOVA) with experimental condition as the grouping/factor variable and pretest score as covariate also indicated that there was significant difference (F(1, 20)=5.093, p=0.035) in mean learning gains between the Prediction First and Self-Explanation First groups while adjusting for pretest scores (prior knowledge). The effect size is small (0.20). The order of the self explanation can explain 20% of variance in learning gain.The better performance of the former group can be attributed to the Self-Explanation learning strategy. The participants in Self-Explanation group first self-explained four tasks before working on prediction tasks. The use of self-explanations had both improved their learning and comprehension of source code with positive effects on the prediction tasks. On the other hand, the Prediction First group only used the self-explanation strategy after they finished working on the prediction tasks.

Is there a relationship between the amount of self-explanation generated and learning gains? For this question, we first analyzed the relationship between count of content words (the most informative words such as nouns, verbs, adjectives, and adverbs) and learning gains. The scatter plot in figure 1 indicates a positive relationship between content word count and learning gains. The relationship between self-explanation (as measured by content word count) and learning gains was further investigated using a Pearson product-moment correlation coefficient. We found a strong, positive correlation between them, r = 0.62, n = 23, p = 0.001, with higher count of content words associated with higher learning gains. Content word production helps to explain nearly 38 per cent of the variance in learning gains.

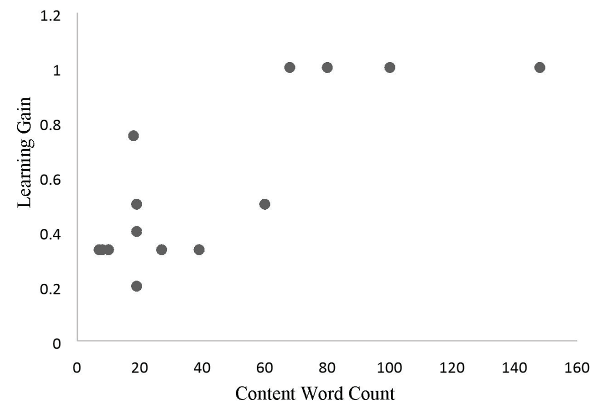

Figure 1: Scatter plot for content word and learning gain.

Group	N	Mean	SD	t-val	Sig.
High Prior	11	0.36	0.58	0.49	0.63
Low Prior	12	0.27	0.21		

Table 3: Independent sample t-test result between the learning gains of high vs. low prior knowledge groups.

Is self-explanations biased towards students prior knowledge? We divided the participants into two groups, high and low prior knowledge, using the mean pretest score of 3.3 as the cut-off value. Table 3 shows that the difference in learning gain is not statistically significant for the groups; thus, we did not found any such evidence of discrepancy of the strategy based on student's prior knowledge.

Nature of Self-Explanation

We found that nature of self-explanation in program comprehension varies a lot in terms of amount of self-explanation generated by participants and how they talk about the code examples shown. As indicated in table 4, participants on average generated 3.52 sentences, 61.35 words, and 34.52 content words per self-explanation task. The amount of self-explanation generated varies a lot; while some participants did not self-explain at all (0 sent count - they simply predicted the output), others wrote 17 sentences, 148 content words and 296 word.

Indeed, self-explanations vary widely. For instance, the first self-explanation in Table5 in row 1 is very detailed and long as opposed to the one shown in the row 4 which simply explains the execution of the code using a mathematical expression. In the latter case, the student succinctly indicates that time is equal to 10 and therefore less than 12 which in turn (this is implied) should lead to the execution of a certain branch of the if statement in the code. In other words, the explanation is correct but incomplete, if only the explicit parts should be accounted for. However, since the student correctly predicted the output the implied parts can be assumed to be correctly considered (but not articulated verbally) by the student.

	Word Count	Content Word Count	Sent Count
Mean	61.35	34.52	3.52
SD	74.2	37.92	4.48
Max	296	148	17
Min	0	0	0

Table 4: Summary statistics of self-explanation per task (all self-explanations from all participants).

ID	Self-Explanation Examples
1	*Factorial of 4 is 24. In main we see that num is 4. We also have a method named factorial that takes in an int number and multiplies it by itself minus one again and again until n is one. That number that is multiplied is what factorial returns. In our case we sent to it the number 4. 4is not 1, so 4 is multiplied by a number that is returned from factorial. So, 4multiplied by (4-1) multiplied by (3-1) multiplied by 2-1 multiplied by 1.Going backwards, 1 multiplied by 2 multiplied by 3 multiplied by 4 results in24*
2	*The for loop keeps iterating as we add one to i until i is greater than 10. Once it is, we exit and print out the result of sum divided by i which is 64 divided by 11 which is 5 (because the variables are ints)*
3	*n(n 1)/2 = in this case, 55, and we divide it by 10,the last value of i. 55/10 is 5.5, but since these are ints, the decimal is truncated off and we are left*
4	*time=10<12*
5	*0 plus 1 plus 2 plus 3 plus 4... plus 10 equals 45.int 45 divided by int 11 is int 4.*

Table 5: Examples of self-explanations.

The second explanation and the fifth indicate students who understand the code but make simple computation errors and therefore incorrectly predict the output. Both students compute the sum of the integers from 1 to 10 being 64 and 45, respectively, instead of the correct value of 55, even though semantically they understand the purpose of the loop. The two explanations, which are about the same code example, reveal two different ways to explain loops. Explanation 2 in Table 2 describe the loop succinctly while explanation 5 describes the loop in its "unwrapped" or sequence-of-iterations form.

Explanation 3 is also about the same code example as explanations 2 and 5 and this is a clear example of students not paying close attention to the code. The value of the iteration variable i is 11 after the loop ends and not 10 (the loop condition is $i <= 10$). While the student understands that the main idea of the loop is to add numbers, it does not pay attention to the details of the loop.

All shown explanations in Table 5 and in fact all 92 explanations we collected from the 23 students (4 explanation tasks per participants), exhibit a combination of language, code references, and mathematical expressions. This has significant implications for any automated methods to assess such self-explanations and therefore any intelligent tutoring system that uses scaffolded self-explanation as a key instructional strategy. Such systems need to automatically evaluate self-explanations in order to provide adequate feedback. Novel assessment methods will need to be developed as we are not aware of any such methods to automatically evaluate self-explanations generated in the context of source code comprehension activities like those shown in Table 5.

Discussion and Future Work

Self-explanation is a promising learning strategy that could help students improve their source code comprehension skills and learn complex programming concepts as our preliminary work here has demonstrated. Indeed, our experiments with open self-explanation prompting provides support for the positive role of self-explanation for source code comprehension and learning. We also found no evidence of biases of the strategy with student's prior knowledge. Like(Recker and Pirolli 1990; Pirolli and Recker 1994; Bielaczyc, Pirolli, and Brown 1995), we also found that learning gains is strongly correlated with the amount of generated self-explanation. We also found that learners self-explain in different ways in terms of the amount of details they provide. Furthermore, the self-explanations include references to code, mathematical expressions, and natural language which poses new challenges for automatically assessing these self-explanations.

Because of small sample size(n=23) and sample population from Asia, the finding of this study may not be generalized to student in USA or other part of the world. Students from two continent might perceive knowledge under same learning strategy differently; which is not taken consideration during this. Furthermore, this study only covered few basic concepts of JAVA such as operator precedence, nested $if - else$, for loops, $while$ loops, arrays, creating objects and using their methods; thus, the study should be further investigated in the context of more examples and broader topics if we are to claim about entire JAVA learning using self explanations.

There are a number of future work directions we envision. First of all, a study that would compare spoken versus typed self-explanations for source code comprehension task would be useful to investigate whether there are any relevant difference in terms of comprehension and learning of computer programming concepts. While such studies have been performed for regular texts, there are no such studies reported for source code comprehension. Furthermore, a source code reading skill assessment instrument is needed in order to assess learners' source code comprehension level. We also plan to run an experiment when students are shown code examples one line at a time and prompted to explain their thoughts at any new line of code. Exploring all these new directions in the future will further reveal details about source code comprehension processes as well as create necessary instruments to assess code comprehension skills. Finally, we plan to annotate self explanations and develop machine learning model for automatic grading of self-explanations. Self explanations in the context of programming varies a

lot,so, it is not possible to compare them with reference answer and simply apply similarity method (Maharjan et al. 2017); thus, seeks novel approach of assessment.

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'. We also like to thank Bikash Balami, Allen Nembang and Bijay Rai for support during experiment.

References

Alhassan, R. 2017. The effect of employing self-explanation strategy with worked examples on acquiring computer programing skills. *Journal of Education and Practice* 8(6):186–196.

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

Best, R. M.; Rowe, M.; Ozuru, Y.; and McNamara, D. S. 2005. Deep-level comprehension of science texts: The role of the reader and the text. *Topics in Language Disorders* 25(1):65–83.

Bielaczyc, K.; Pirolli, P. L.; and Brown, A. L. 1995. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and instruction* 13(2):221–252.

Chi, M. T.; De Leeuw, N.; Chiu, M.-H.; and LaVancher, C. 1994. Eliciting self-explanations improves understanding. *Cognitive science* 18(3):439–477.

Crippen, K. J., and Earl, B. L. 2004. Considering the efficacy of web-based worked examples in introductory chemistry. *Journal of Computers in Mathematics and Science Teaching* 23(2):151–167.

Du Boulay, B. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2(1):57–73.

Guzdial, M., and Soloway, E. 2002. Teaching the nintendo generation to program. *Communications of the ACM* 45(4):17–21.

Hung, Y.-C. 2012. Combining self-explaining with computer architecture diagrams to enhance the learning of assembly language programming. *IEEE Transactions on Education* 55(4):546–551.

Kwon, K., and Jonassen, D. H. 2011. The influence of reflective self-explanations on problem-solving performance. *Journal of Educational Computing Research* 44(3):247–263.

Kwon, K.; Kumalasari, C. D.; and Howland, J. L. 2011. Self-explanation prompts on problem-solving performance in an interactive learning environment. *Journal of Interactive Online Learning* 10(2).

Maharjan, N.; Banjade, R.; Gautam, D.; Tamang, L. J.; and Rus, V. 2017. Dt_team at semeval-2017 task 1: Semantic similarity using alignments, sentence-level embeddings and gaussian mixture model output. In *Proceedings of the 11th international workshop on semantic evaluation (semeval-2017)*, 120–124.

Marx, J. D., and Cummings, K. 2007. Normalized change. *American Journal of Physics* 75(1):87–91.

McNamara, D. S., and Magliano, J. P. 2009. Self-explanation and metacognition: The dynamics of reading. In *Handbook of metacognition in education*. Routledge. 72–94.

McNamara, D. S. 2009. The importance of teaching reading strategies. *Perspectives on language and literacy* 35(2):34.

Morrison, B. B.; Margulieux, L. E.; and Guzdial, M. 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. ACM.

Muñoz, B.; Magliano, J. P.; Sheridan, R.; and McNamara, D. S. 2006. Typing versus thinking aloud when reading: Implications for computer-based assessment and training tools. *Behavior research methods* 38(2):211–217.

Pirolli, P., and Recker, M. 1994. Learning strategies and transfer in the domain of programming. *Cognition and instruction* 12(3):235–275.

Ramalingam, V.; LaBelle, D.; and Wiedenbeck, S. 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.

Recker, M. M., and Pirolli, P. 1990. A model of self-explanation strategies of instructional text and examples in the acquisition of programming skills.

Rezel, E. S. 2003. The effect of training subjects in self-explanation strategies on problem solving success in computer programming.

Roy, M., and Chi, M. T. 2005. The self-explanation principle in multimedia learning. *The Cambridge handbook of multimedia learning* 271–286.

Rus, V.; Brusilovsky, P.; Fleming, S.; Tamang, L.; Akhuseyinoglu, K.; Barria-Pineda, J.; Ait-Khayi, N.; and Alshaikh, Z. 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*.

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

Van Merriënboer, J. J., and Sluijsmans, D. M. 2009. Toward a synthesis of cognitive load theory, four-component instructional design, and self-directed learning. *Educational Psychology Review* 21(1):55–66.

Wilson, C., S. L. A. S. C. . S. M. 2010. Running on empty: The failure to teach k-12 computer science in the digital age. *New York, NY: The Association for Computing Machinery and the Computer Science Teachers Association*.