Scaffolding Support for Self-Explaining Worked Examples in Learning Computer Programming

Chun-Ying Chen[1], Shih-Hsuan Wei[2], Pey-Ru Yen[3]
[1]*Center for General Education*
[2]*Department of Mathematics Education*
[3]*Department of Education*
National Taichung University of Education
Taiwan

Abstract

Research has shown that novices benefit most from worked examples when they actively engage with them through self-explanations. However, learners often need prompts to self-explain, as they may not do so spontaneously. This study examined how scaffolding support—specifically, guided questions and a subgoal-labeled flowchart—can facilitate effective self-explanations in learning computer programming. The experiment compared three instructional conditions: (1) worked examples with instructional explanations, (2) worked examples with guided questions prompting self-explanations, and (3) worked examples with a subgoal-labeled flowchart prompting self-explanations. Active self-explanations were those supported by either guided questions or a subgoal-labeled flowchart. We evaluated participants' cognitive load during learning and assessed learning outcomes using a recall posttest and a transfer task of programming problem-solving. The findings suggest that subgoal-labeled flowcharts were an effective scaffolded support to enhance novices' transfer performance.

1. Introduction

Learning to program is a complex task, particularly for novices. To support this process, programming courses often employ example-based approaches. Worked examples—fully worked-out problem solutions—can reduce cognitive load and have been shown to be more effective than traditional practice-based problem-solving [18], [21]. Research further suggests that novices benefit most when they actively explain each step of worked examples to themselves, rather than reviewing them passively (e.g., [7], [9]). Such active self-explanations promote deeper cognitive engagement, thereby enhancing learning and supporting knowledge construction [27].

This study investigated how to effectively prompt self-explanations in the context of learning computer programming. Two types of scaffolding support were examined. The first involved guided questions designed to help learners consider how to implement the solution in code, directing attention to relevant conditions and encouraging reflective engagement. The second type of scaffolding was a flowchart integrated with subgoal labels [5] that represent conceptually meaningful chunks of a problem's solution or subgoals by explicitly labeling each solution step. This approach was intended not only to reduce cognitive load but also to scaffold the problem-solving process. The aim of the study was to determine how these scaffolds could facilitate effective self-explanations and, in turn, enhance learning outcomes.

2. Theoretical Background and Related Work

This section outlines the theoretical and empirical foundations of the study, focusing on cognitive theories, the benefits of worked examples and self-explanations, their applications in programming education, and the role of scaffolding tools like guided questions and subgoal-labeled flowcharts in supporting novices' learning.

2.1. Theoretical Foundation

The theoretical foundation of this study is Cognitive Load Theory (CLT; [21], [22]). CLT emphasizes the major limitations of working memory in terms of both capacity and duration as key constraints in learning.

According to CLT, the cognitive load arises from three sources: intrinsic, extraneous, and germane load (IL, EL, and GL). IL reflects the inherent complexity of the learning material relative to the learner's prior knowledge. A central determinant of IL is element interactivity — the number of information elements that must be simultaneously processed in working memory [23]. EL refers to unnecessary cognitive effort caused by inadequate instructional design and that does not contribute to learning. GL, in contrast,

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

denotes the mental effort devoted to processes that foster meaningful learning, such as schema construction and integration, thereby reflecting the active engagement of working memory resources [23]. Overall, CLT emphasizes minimizing unnecessary EL while maximizing GL, all within the limited capacity of working memory.

2.2. The Worked Examples and Self-Explanations

Novices often experience high IL from learning materials due to limited prior knowledge. They learn better from the instructional format, such as worked examples (i.e., fully worked-out solutions) than from practice problems they must solve alone. When provided with worked examples before problem-solving, novices typically adopt more efficient strategies and attend to the structural features of problems. In contrast, when faced with traditional practice exercises, they frequently rely on trial-and-error strategies. Worked examples thus serve as a valuable means for novices to acquire schemata by studying expert solutions (see [18], [21]).

Self-explanation is a constructive learning activity in which learners generate verbal statements, often involving inferences, to clarify the meaning of the material for themselves [8]. Through self-explaining, learners integrate new information with prior knowledge and derive new insights [27]. Self-explanation prompts can take several forms, including open-ended, focused, scaffolded, resource-based, and menu-based formats [27]. Research indicates that more structured prompts—such as focused, scaffolded, or resource-based ones—lead to deeper learning than open-ended prompts [27]. For novices in particular, guidance and scaffolding help direct attention to relevant aspects of the material, making self-explanation more effective. Moreover, self-explanation has been shown to be generally more beneficial for learning than instructional explanation provided by instructors [2], [12].

2.3. Self-Explaining Worked Examples in Programming Education

Learners' self-explanations of worked examples showed benefits for learning in computer programming [14], [16], [25]. Prior studies show several approaches to elicit self-explanations, such as providing reflection prompts, questioning strategies, writing in-code comments, and using subgoal labels. In learning JavaScript, students performed better in problem-solving tasks when actively engaged in the reflective self-explanation process [10]. Similarly, when studying Java, researchers compared three groups: a control group with no self-explanation activity, a group asked to self-explain, and a group provided with multiple-choice questions designed to direct attention to specific code features. Results indicated that self-explanation improved performance, with additional gains observed when guiding questions helped learners focus their explanations [26].

Using in-code comments as a self-explanation strategy for students' engagement with the worked examples, Vieira et al. [24] identified students perceived uses of the in-code commenting activities, including understanding the example, making a connection between the programming code and the disciplinary problem, and becoming familiar with the programming constructs. When teaching programming C# and algorithm design, Vieira et al. [25] show that students can effectively carry out an elaborated self-explanation (in-code comments) of their coded solutions when using worked examples pedagogy, leading to improved learning outcomes. However, students also noted that excessive or overly detailed comments reduced code readability and could increase cognitive load.

Garces et al. [9] examined the integration of debugging and self-explaining when teaching C programming. They concluded that sequencing these activities—self-explaining first, debugging second—may be particularly effective. Likewise, in teaching application development with App Inventor, learner-constructed subgoal labels were found to scaffold problem-solving by requiring students to break down procedures into subgoals and explain the purpose of each solution step [12]. This strategy was designed to strengthen self-explanations while reducing the likelihood of learners producing ineffective or inaccurate ones.

2.4. Scaffolding for Computer Programming with Subgoal-Labeled Flowcharts

Programming education is crucial in developing computational thinking, a problem-solving approach that involves decomposition, abstraction, algorithm development, and evaluation. Given the close relationship between computing and programming, computational thinking skills are often enhanced through computational tools such as programming languages and environments [19]. Because the cognitive process is quite abstract and complex due to the nature of programming problem-solving, novices easily get overloaded when they have to juggle programming constructs and problem-solving processes simultaneously. It often overwhelms novice programmers while confronting many interacting elements to be learned at once [21]. Besides the worked examples, programming flowcharts were often used as learning scaffolding to support novice programmers (e.g., [3], [20], [28]).

A programming flowchart is a visual tool that programmers use to understand the problem-solving process or algorithm when creating applications. The

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

flowchart typically uses geometric shapes to represent steps and arrows to communicate the flow of data and the flow of execution of a program, so this tool offers guidance for explicit problem-solving processes for novices. In this regard, presenting a matched flowchart representing the visualized solution of the worked example was assumed to mitigate novices' cognitive load during learning.

Subgoal learning theory [5] further enhances this support by encouraging learners to organize procedures into meaningful, goal-oriented units. Subgoal labels—brief descriptors that group related steps based on their function—assist learners in identifying the structure and purpose of different parts of a procedure. Prior research supports the use of subgoal-labeled examples to enhance learning. For instance, Margulieux et al. [4] found that learners exposed to subgoal-labeled worked examples demonstrated improved performance and greater transfer to new problems. Similarly, Margulieux and Catrambone [12] showed that students who generated their own subgoal explanations outperformed peers on problem-solving tasks.

3. The Experiment Design

This study investigated the effects of scaffolding support to prompt effective self-explanations with worked examples for learning computer programming on learners' cognitive load and problem-solving performance. We proposed two types of scaffolding support: guided questions and a subgoal-labeled flowchart.

This experiment included three conditions: (1) worked examples with instructional explanations (IE), (2) worked examples with guided questions prompting self-explanations (QE), and (3) worked examples with a subgoal-labeled flowchart prompting self-explanations (FE). The IE condition was the control group. There were two examples and problems in each condition to ensure transfer performance, as suggested in previous research [17]. Each example was followed directly by a similar problem to solve.

Learners engaging in self-explanations of the examples could ensure the worked example effect. Instructional explanations are generally regarded as the most cognitively passive form of support [27]. In contrast, active self-explanations can be fostered through scaffolds such as guided questions or subgoal-labeled flowcharts. Guided questions prompt learners to reflect on how to code the solution to a problem, thereby directing attention to relevant aspects of the task and potentially inducing higher GL for understanding and self-explaining. A subgoal-labeled flowchart, by explicitly labeling each solution step and grouping them into meaningful chunks, highlights the structure of the problem's solution. This design encourages learners to actively engage with worked examples, promoting deeper understanding. Accordingly, we expected the flowchart to reduce EL and increase GL, ultimately enhancing student learning.

The discussions above lead to the development of the following hypotheses:

*Hypothesis 1***.** Participants in the IE condition would exhibit lower GL and lower transfer performance than those in other conditions.

Hypothesis 2. Participants in the QE condition would exhibit higher GL and higher transfer performance than those in the IE condition.

Hypothesis 3. Participants in the FE condition would exhibit lower EL, higher GL, and higher transfer performance than those in other conditions.

4. Method

This section details the study's methodology, including participant recruitment, learning materials, assessment measures, and experimental procedures.

4.1. Participants

The study involved 98 university students from two regular classes during the Spring term at the Center of General Education in Taiwan. The experiment was integrated into their regular classes, which were introductory courses in computational thinking and programming tailored for non-computer majors. Students from various disciplines, except those in computer-related fields, were eligible to participate, with the majority coming from Humanities and Social Sciences. To support novice learners, Scratch—a block-based programming tool—has been selected as the learning environment. Unlike text-based programming languages, its highly visual design could help minimize common beginner errors such as syntax and typing mistakes.

At the beginning of the semester, all participants were surveyed to assess their programming experiences and asked to take the pretest to evaluate their prior knowledge of Scratch. All students participated in the learning tasks of the experiment as part of their regular class activities. At the end of the semester, data were collected from those with no prior experience with Scratch or any programming.

4.2. Learning Materials

The instructor (the first author) initially taught computational concepts using all block functions of Scratch, covering data representation with variables and lists, flow controls, abstraction and problem decomposition, user interactivity, parallelism, and synchronization. These foundational lessons spanned eight weeks. The class then applied what they have learned to create Scratch projects focused on programming problem-solving.

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

By adapting original examples from Marji [13], we designed and developed all formats of worked examples for programming projects involving the classification of different types of triangles and quadrilaterals. Each condition included two worked examples: one for a triangle classification game and one for a quadrilateral classification game. The first programming task involved creating a game to identify different types of triangles. The second task required programming a game to identify different kinds of quadrilaterals.

To ensure participants could solve the problems by analogy, the two worked examples differed only in contextual/surface features, while the procedural steps remained similar. The two worked examples in the IE condition included in-code comments written by the instructor. The other two worked examples in the QE and FE conditions did not include in-code comments because participants were required to complete that portion as part of the assignment. The two worked examples in the QE condition included eight guided questions to prompt self-explanations (writing in-code comments). The two worked examples in the FE condition included a matched flowchart integrated with the same subgoal labels (see Figure 1).

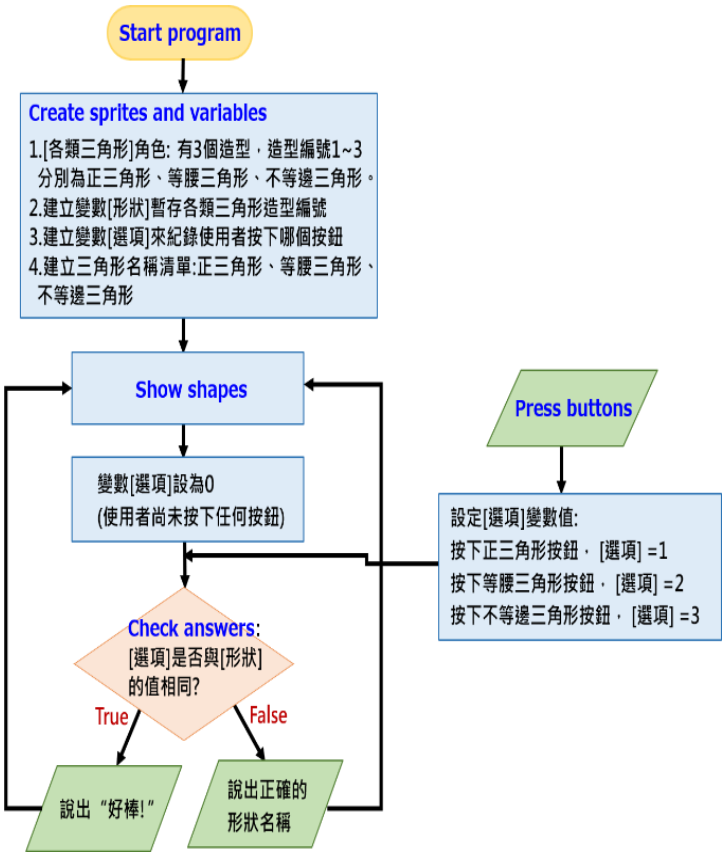

Figure 1. A subgoal-labeled flowchart

The subgoal labels in bold font included "start program," "create sprites and variables," "show shapes," "press buttons," and "check answers."

4.3. Measures

All measures were web-based assessments and consisted of (a) a pretest that evaluated participants' prior knowledge, (b) a posttest and a programming task that measured the learning outcomes, and (c) cognitive load ratings that evaluated participants' experienced cognitive load during learning.

Both pretest and posttest were composed of ten multiple-choice questions that were designed to test the basic computational concepts of Scratch programming. The test items for both tests were slightly modified (i.e., changing variable names, values, or sizes) to minimize the testing effect. The max time allowed for completing the test was 10 min. One point was awarded for every correct answer, with a max score of 10.

The transfer task was a programming activity. Participants were required to program a game to identify different shapes, including rectangles, parallelograms, triangles, trapezoids, circles, and sectors. They received a problem description and a sample run of the program and were tasked with constructing a complete program to solve the

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

problem. The SVG graphics required for the program sprites were provided in advance, and participants needed to code each sprite. The program contained seven sprites: six for the answer buttons and one named Shapes, which contained the main script. The main script was awarded a total of 40 points if correctly coded. The other six sprites were awarded 10 points each if correctly coded. Thus, the max score for the correctness of the constructed programs was 100 points. Participants had a max of 40 min to complete this task. The subjective cognitive load rating scale (see Table 1) was adapted from Leppink et al. [11] and Morrison et al. [15]. Cognitive load during the learning phase was measured with 13 items assessing IL (items 1-4), EL (items 5-8), and GL (items 9-13). Participants rated the items on a 10-point scale, with zero indicating *not at all the case* and 10 indicating *completely the case*.

Table 1. Cognitive load measurement

Type	Corresponding items
Intrinsic	1. The content of this lesson was very complex. 2. The lesson covered program code that I perceived as very complex. 3. The lesson covered concepts and definitions that I perceived as very complex. 4. I invested great mental effort because of the complexity of this lesson.
Extraneous	5. The instructions and explanations during the lesson were very unclear. 6. The instructions and explanations during the lesson were full of unclear language. 7. The instructions and explanations during the lesson were, in terms of learning, very ineffective. 8. I invested great mental effort on unclear and ineffective instructions and explanations during the lesson.
Germane	9. This lesson really enhanced my understanding of the content covered. 10. This lesson really enhanced my knowledge and understanding of computing/programming. 11. This lesson really enhanced my understanding of the program code covered. 12. This lesson really enhanced my understanding of the concepts and definitions. 13. I invested great mental effort during this lesson to enhance my knowledge and understanding.

Note. Adapted from Leppink et al. [11] and Morrison et al. [15].

4.4. Procedures

The experiment was conducted in the ninth week of the semester when participants began to learn programming problem-solving by creating Scratch projects. Tallied with the course syllabus of the two classes, the experiment was conducted in two sessions in the computer lab where participants normally attended computer-based classes. Participants were randomly assigned to one of the three conditions in each session, with each participant working independently on a computer.

The experiment consisted of an example-based learning phase followed by test phases. Initially, participants received brief instructions about the experiment's procedure. During the learning phase, participants were not allowed to leave the classroom or talk to others except to ask the instructor questions if they encountered problems. Time-on-task was restricted in both the learning and test phases. Participants were given 25 min to complete each example-based learning task. Until time expired, they were encouraged to review the learning materials or explore the Scratch programming environment. Following the learning phase, participants had 10 min to complete the cognitive load assessment. Finally, they were given 10 min to complete the posttest and 40 min to complete the programming task.

5. Results

The data were analyzed with one-way analyses of variance (ANOVA). There were three conditions for comparisons. The dependent variables were learning outcomes regarding the recall posttest score and a transfer task of programming problem-solving, and different types of cognitive load experienced during the learning phase. Post-hoc tests were conducted as follow-up analyses when there was a significant F test. Alpha was set at 0.05 for all statistical tests. Effect sizes were reported using eta-squared (η^2), with values of .01, .06, and .14 indicating small, medium, and large effects, respectively [6]. Descriptive statistics are summarized in Table 2. The conditions did not differ in the pretest ($F(2, 95) = 1.602$, $p = .207$), indicating comparable prior knowledge.

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

Table 2. Descriptive statistics for all measures

	IE Group	QE Group	FE Group
Pretest (0-10)	5.61 (1.35)	5.09 (1.42)	5.64 (1.34)
Recall (0-10)	8.73 (1.40)	9.03 (1.26)	9.30 (0.98)
Transfer (0-100)	79.39 (20.31)	80.25 (14.77)	90.88 (9.20)
Cognitive load during learning			
Intrinsic (0−10)	7.74 (1.19)	6.92 (1.76)	6.41 (1.87)
Extraneous (0−10)	6.45 (1.48)	5.23 (1.61)	4.76 (1.91)
Germane (0−10)	6.56 (1.68)	6.96 (1.58)	6.67 (1.79)

5.1. Cognitive Load

ANOVA found significant differences between some of the three conditions for IL ($F(2, 95) = 5.582$, $p = .005$, $\eta 2 = .105$), and EL ($F(2, 95) = 8.979$, $p < .001$, $\eta 2 = .159$); but not for GL ($F(2, 95) = .467$, $p = .628$). For IL, HSD revealed that participants in the FE condition reported significantly lower IL than those in the IE condition ($p = .004$). However, no significant differences in IL were found between the IE and QE conditions ($p = .112$) or between the QE and FE conditions ($p = .419$). For EL, HSD revealed significant differences between the IE and QE conditions ($p = .012$), as well as between the IE and FE conditions ($p < .001$). The results indicated that participants in both the QE and FE conditions reported significantly lower EL than those in the IE condition. However, no significant difference in EL was found between the QE and FE conditions ($p = .488$).

5.2. Learning Outcomes

ANOVA found significant differences between some of the three conditions in transfer performance ($F(2, 95) = 5.638$, $p = .005$, $\eta 2 = .106$); but not in recall performance ($F(2, 95) = 1.825$, $p = .167$). HSD revealed that participants in the FE condition outperformed those in both the IE and QE conditions ($p = .009$ and $.018$, respectively) for transfer performance. However, no significant difference was found between the IE and QE conditions for transfer performance ($p = .973$).

6. Discussion

This study examined the effects of two scaffolding supports—guided questions and subgoal-labeled flowcharts—on novice learners' self-explanations with worked examples, focusing on cognitive load and transfer performance in programming problem-solving. Hypothesis 1 predicted that participants in the IE condition would exhibit lower GL and lower transfer performance than those in other conditions. This was only partially supported. Unexpectedly, no significant differences in GL were found across the three conditions. However, participants in the FE condition outperformed those in the IE condition on transfer performance, whereas those in the QE condition did not show superior transfer performance compared to those in the IE condition.

Hypothesis 2 predicted that participants in the QE condition would exhibit higher GL and higher transfer performance than those in the IE condition. Contrary to expectations, this was not supported. No significant differences were found in GL or transfer performance between the two groups. Interestingly, participants in the QE condition reported significantly lower EL than those in the IE condition, though this reduction in EL did not translate into better transfer performance. These results align with prior findings showing no significant performance differences between guided-question and control groups, even though students perceived the prompts as helpful for remembering content and monitoring understanding [1].

Hypothesis 3 predicted that participants in the FE condition would exhibit lower EL, higher GL, and higher transfer performance than those in other conditions. This was partially supported. Participants in the FE condition reported significantly lower IL than those in the IE condition only, but not significantly lower than those in the QE condition. As expected, they reported significantly lower EL than both other conditions. However, GL did not differ across conditions. Despite this, the FE condition consistently led to superior transfer performance relative to the other groups.

7. Limitations

This study has several limitations. First, it was conducted in the context of an introductory programming course using a block-based visual

 | DOI: 10.20533/ijicr.2042.4655.2025.0166

programming environment focused on geometric shape classification tasks. As such, the findings may not generalize to more advanced programming contexts or text-based languages such as Python. Future research should examine whether similar scaffolding benefits extend to more complex, syntax-driven environments. Second, the experiment was carried out in a controlled classroom setting with strict time constraints, which may not fully reflect authentic learning conditions where learners progress at their own pace. The imposed structure may have limited participants' opportunities to reflect more deeply or revise their explanations (i.e., in-code comments).

Finally, GL was assessed using self-report measures, which are known to have limitations in capturing actual cognitive engagement. Future work should incorporate more objective or behavioral indicators of germane processing to strengthen the validity of cognitive load assessments.

8. Conclusion

In conclusion, the findings demonstrate that both scaffolding approaches effectively reduced EL, but subgoal-labeled flowcharts offered additional benefits by lowering IL and improving transfer performance. Guided questions, while successful in reducing EL, did not translate into measurable gains in transfer. Notably, no significant differences in GL were observed, consistent with concerns about the methodological limitations of self-report measures of GL [11], [23]. Future research should refine scaffold designs and adopt alternative measures of cognitive engagement—such as motivational and behavioral indicators—to more accurately assess GL in programming and other complex learning domains.

9. References

[1] Aureliano, V.C., Patricia, C., and Caspersen, M.E. (2016). Learning programming through stepwise self-explanations. In 2016 11th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1-6). IEEE.

[2] Bisra, K., Liu, Q., Nesbit, J.C., Salimi, F., and Winne, P.H. (2018). Inducing self-explanation: A meta-analysis. Educational Psychology Review, 30, 703-725.

[3] Cabo, C. (2018, October). Effectiveness of flowcharting as a scaffolding tool to learn python. In 2018 IEEE Frontiers in Education Conference (FIE) (pp. 1-7). IEEE.

[4] Margulieux, L.E., Catrambone, R., and Guzdial, M.

(2016). Employing subgoals in computer programming education. Computer Science Education, 26(1), 44-67. DOI: 10.1080/08993408.2016.1144429

[5] Catrambone, R. (1998). The subgoal learning model: Creating better examples so that students can solve novel problems. Journal of Experimental Psychology: General, 127, 355–376.

[6] Cohen, J. (1988). Statistical power analysis for the behavioral sciences (2nd ed.). Hillsdale, NJ: Erlbaum.

[7] Crippen, K.J., and Earl, B.L. (2007). The impact of web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. Computers and Education, 49(3), 809-821.

[8] Fiorella, L., and Mayer, R.E. (2022). The Generative Activity Principle in Multimedia Learning. The Cambridge Handbook of Multimedia Learning, 339-350.

[9] Garces, S., Vieira, C., Ravai, G., and Magana, A.J. (2023). Engaging students in active exploration of programming worked examples. Education and Information Technologies, 28(3), 2869-2886.

[10] 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.

[11] Leppink, J., Paas, F., van Gog, T., van der Vleuten, C.P.M., and van Merriënboer, J.J.G. (2014). Effects of pairs of problems and examples on task performance and different types of cognitive load. Learning and Instruction, 30, 32-42.

[12] Margulieux, L.E., and Catrambone, R. (2021). Scaffolding problem solving with learners' own self explanations of subgoals. Journal of Computing in Higher Education, 33, 499-523.

[13] Marji, M. (2014). Learn to program with Scratch: A visual introduction to programming with games, art, science, and math. No Starch Press.

[14] Sandoval-Medina, C., Arévalo-Mercado, C.A., Muñoz-Andrade, E.L., and Muñoz-Arteaga, J. (2024). Self-explanation effect of cognitive load theory in teaching basic programming. Journal of Information Systems Education, 35(3), 303-312. DOI: 10.62273/29]GMIV1698.

[15] Morrison, B.B., Dorn, B., and Guzdial, M. (2014, July). Measuring cognitive load in introductory CS: Adaptation of an instrument. In Proceedings of the tenth annual conference on international computing education research (pp. 131-138).

[16] Pirolli, P., and Recker, M. (1994). Learning strategies and transfer in the domain of programming. Cognition and Instruction, 235-275.

[17] Reed, S.K., and Bolstad, C.A. (1991). Use of examples and procedures in problem solving. Journal of Experimental Psychology: Learning, Memory, and Cognition, 17(4), 753.

[18] Renkl, A. (2021). The worked example principle in multimedia learning. In R.E. Mayer, and L. Fiorella (Eds.), The Cambridge handbook of multimedia learning (3rd ed., pp. 231-240). Cambridge University Press.

[19] Shute, V. J., Sun, C., and Asbell-Clarke, J. (2017).

Demystifying computational thinking. Educational Research Review, 22, 142-158.

[20] Smetsers-Weeda, R., and Smetsers, S. (2017, November). Problem solving and algorithmic development with flowcharts. In Proceedings of the 12th Workshop on Primary and Secondary Computing Education, 25-34.

[21] Sweller, J., Ayres, P., and Kalyuga, S. (2011). Cognitive load theory. New York: Springer.

[22] Sweller, J., van Merriënboer, J., and Paas, F. (2019). Cognitive architecture and instructional design: 20 years later. Educational Psychology Review, 31, 261–292.

[23] Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. Educational psychology review, 22, 123-138. DOI: 10.1007/s10648-010-9128-5.

[24] Vieira, C., Magana, A.J., Falk, M.L., and Garcia, R.E. (2017). Writing in-code comments to self-explain in computational science and engineering education. ACM Transactions on Computing Education (TOCE), 17(4), 1-21.

[25] Vieira, C., Yan, J., and Magana, A.J. (2015). Exploring design characteristics of worked examples to support programming and algorithm design. Journal of Computational Science Education, 6(1), 2-15.

[26] Vihavainen, A., Miller, C.S., and Settle, A. (2015, February). Benefits of self-explanation in introductory programming. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 284-289).

[27] Wylie, R., and Chi, M.T.H. (2014). The self-explanation principle in multimedia learning. In R.E. Mayer (Ed.), The Cambridge handbook of multimedia learning (2nd ed., pp. 413-432). Cambridge University Press.

[28] Zhang, J.H., Meng, B., Zou, L.C., Zhu, Y., and Hwang, G.J. (2023). Progressive flowchart development scaffolding to improve university students' computational thinking and programming self-efficacy. Interactive Learning Environments, 31(6), 3792-3809.

Funding

This work was supported by National Science and Technology Council in Taiwan [Grant No. NSTC 113-2635-H-142-001].

Acknowledgment

The authors gratefully acknowledge the opportunity to present this work at the 15th Canada International Conference on Education (CICE-2025), which held from the 22nd to 24th of July 2025 at the Toronto Metropolitan University, Toronto, Canada.

Institutional Review Board Statement: The Research Ethics Committee of National Chung Cheng University, Taiwan has granted approval for this study on 21 November 2024 [App. No. CCUREC113052404].