Collaborative Reflection "in the flow" of Programming: Designing Effective Collaborative Learning Activities in Advanced Computer Science Contexts

Sreecharan Sankaranarayanan, Lanmingqi Ma, Siddharth Reddy Kandimalla, Ihor Markevych, Huy Nguyen, R. Charles Murray, Christopher Bogart, Michael Hilton, Majd Sakr, Carolyn Penstein Rosé
sreechas@cs.cmu.edu, lanmingm@andrew.cmu.edu, skandima@andrew.cmu.edu, imarkevy@andrew.cmu.edu, hn1@andrew.cmu.edu, rcmurray@andrew.cmu.edu, cbogart@andrew.cmu.edu, mhilton@cmu.edu, msakr@cmu.edu, cprose@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract: Designing activities for maximizing collaborative learning in advanced computer science contexts is of broad interest. While programming exercises remain the dominant form of pedagogy here, prior work showed that collaborative reflection over worked examples is as good or even better for conceptual learning and future programming. This work used a "phased" design, with separate collaborative reflection and programming phases, and varied the time boundary between the two to determine their differential impact. A more effective design, however, could involve collaborative reflection prompted "in the flow" of programming, with benefits similar to self-explanation prompts interleaved into individual problem-solving. While total time-on-task is the same, this "interleaved" design might allow learners to spend a larger proportion of this time on reflection. Thus, this paper compares this novel interleaved approach to the phased design. We determine that interleaving increases the proportion of time available for reflection resulting in performance improvements on future programming.

Introduction and Motivation

Advanced courses in the computer science (CS) curriculum aim to help students learn an advanced knowledge area such as machine learning, data science, or cloud computing. Learning an advanced knowledge area requires students to learn its conceptual aspects, as well as the procedural aspects of applying those concepts, once learned, into computer programs. CS pedagogy, however, is dominated by programming exercises, which are believed to be the best way for students to learn CS. These exercises require hands-on implementation that demonstrate procedural knowledge, but only implicitly support conceptual knowledge acquisition without explicit scaffolding.

Although these individual, hands-on programming exercises are the dominant form of engaging with advanced CS content in current practice, the question of whether they are indeed the best way for learners to acquire both conceptual and procedural knowledge has been challenged in prior work (Sankaranarayanan et al., 2020a; Sankaranarayanan et al., 2020b; Sankaranarayanan et al., 2021a; Sankaranarayanan et al., 2021b). One important finding from this work was that learners in advanced CS courses need less help with the procedural aspects of programming (Robins et al., 2003; Garner et al., 2005). They are proficient enough to implement working programs, with little external support once they understand the conceptual aspects. Consequently, their time is better spent on acquiring conceptual knowledge in these advanced knowledge areas.

To best acquire conceptual knowledge, learning science literature, such as the literature on example-based learning, does not prefer problem-solving practice, instantiated here as programming practice. Instead, other types of activities, such as worked example-based reflection, are shown to be more effective for conceptual learning (Renkl, 2014). Prior work showed that this result holds for the advanced CS context as well. Synchronous, collaborative activities predominantly involving collaborative reflection over worked examples were as good as or better, not only for conceptual learning from the task but also for performance on future programming tasks when compared to programming practice alone (Sankaranarayanan et al., 2021a, Sankaranarayanan et al., 2021b). These studies used a "phased" design, as is typical of scripted collaboration (Fischer et al., 2013), to divide the time between collaborative reflection and programming phases using a macro script. The time boundary between the phases is then varied to measure the differential impact of the two on the outcome measures using a between-subjects design. The results showed that collaborative reflection can be as good as, or more effective than programming practice, thus challenging the status quo in CS education pedagogy. Now that it has been established that collaborative reflection can be as good or better than programming practice, the question of improving this benefit by increasing exposure to collaborative reflection opportunities becomes pertinent.

One way to increase exposure to reflection opportunities while keeping total time-on-task constant is by providing them "in the flow" of problem-solving rather than separating the two into their own phases. This "interleaved" design increases the availability of reflection opportunities throughout the task but could also be detrimental since it divides students' attention between reflection and problem-solving as there is no clear boundary between two. Reflection during problem-solving has precedent in the cognitive science literature with the "self-explanation effect". There, a beneficial impact on an individual's learning and performance on transfer tasks is produced from self-explanation prompted during problem-solving (Aleven & Koedinger, 2002). Although those results were in the individual context, it may be possible to hypothesize that collaborative reflection plays a role similar to self-explanation in the collaborative learning context resulting in similar beneficial effects. To test this hypothesis, therefore, a comparison between a design where collaborative reflection is prompted "in the flow" of programming and the phased design from prior work that separates the two into their own phases is warranted.

To design such an activity that provides opportunities for collaborative reflection "in the flow" of programming, we take inspiration from tasks like Parson's Problems in CS education literature (Ericson et al., 2017) that combine elements of reflection and programming practice, and adapt them for use in a collaborative learning context. In our novel activity design, we first present each group with two alternative worked example solutions for each task in the activity in the form of pseudocodes (program outlines). The solutions are designed so that the process of deciding between them draws the group's attention to an important conceptual aspect that they are expected to learn from the task. Conversational agent-based prompts support the group in this endeavor. All the while, the group can implement the solution of their choice going between reflection and programming, as necessary. In the phased design used in prior work, learners first participate in a collaborative reflection phase where they are presented with one worked example pseudocode solution. Instead of an alternative solution, it is the conversational agent-based prompts that present conceptually relevant aspects from an alternative solution for the group to consider. Once the group discusses and decides on a solution to implement, they move on to the programming phase to implement their chosen solution. In both cases, implementation requires translating the pseudocode, which is an algorithmic outline into an actual working program. While the novel activity design and the phased design differ in their "interleaved" and "phased" natures, the same content is provided to learners ensuring that information is controlled across the two conditions in a between-subjects design.

This study tests the hypothesis that interleaving example-based reflection with computer programming presents additional benefits over a phased approach since it increases exposure to reflection opportunities while keeping total time-on-task the same. In a between-subjects design, the interleaved and phased designs are compared to determine differences in conceptual learning from the task as measured by a pre- to post-test, and performance on future programming as measured by performance on hands-on follow-up tasks.

Results show that the interleaved approach maintains the high degree of conceptual learning that the phased approach has while improving performance on hands-on follow-up tasks. Therefore, through the novel, interleaved design, we amplify the benefit to future programming performance. In the following sections, we first describe the theoretical foundations for all the components of this study. We then describe our experimental methods including the context in which the study was conducted, the design of the collaborative learning activity overall, the number and kind of participants/groups, and the conditions in the between-subjects design with examples. Following this, we describe the analysis and results, takeaways from them and future directions.

Theoretical Foundations

This work draws on theory from CS education research, learning science research including cognitive load theory (Sweller et al., 1998) and example-based learning (Renkl, 2014), cognitive science research with the self-explanation effect (Aleven & Koedinger, 2002), and the script theory of guidance (Fischer et al., 2013) for the actual design of the collaborative learning activities.

Computer Science Education Research

Designing for learners in an advanced computer science (CS) context requires an understanding of what "learning" CS actually means. Early research in CS pedagogy (Soloway & Elrich, 1984; Kant & Newell, 1984; Soloway, 1986) had already started to tease apart programming knowledge into the syntax and semantics of programming language constructs, and the mechanisms and explanations used to compose solutions. In other words, students learning programming have to learn the conceptual aspects such as variables, loops, iteration, and functions, in addition to the procedural aspects of turning these concepts into working code. Later research built on this knowledge separation to determine, through an analysis of the differences between novice and expert programmers, that novices need a lot more support with the procedural aspects of programming, while the experts are able to implement concepts, once they are learned, with little external support (Robins et al., 2003; Garner et al., 2005). Consequently, at places in the CS curriculum where students have gained enough procedural knowledge to implement concepts once they are learned, it becomes possible to surmise that their time is better spent learning those concepts instead of superfluously spending time on the procedures.

Owing either to inertia, or the strongly held belief of programming practice being necessary for learning CS, prevailing pedagogical practices have overwhelmingly focused on individual programming exercises (Kalyuga et al., 2001). Consequently, research about alternative activity designs that explicitly target procedural and conceptual knowledge in CS contexts have only started to emerge recently. Two-dimensional Parson's problems (Ericson et al., 2017), a type of code completion problem useful for teaching syntactic and semantic language constructs i.e., the procedural aspects of programming, are one example. Worked example study (Margulieux et al., 2019) targeting the conceptual aspects is another. Even so, these alternative methods of engagement are largely absent as students move from the introductory context on to advanced courses in the computer science curriculum. To design for the advanced context, where a learner's time is better spent on conceptual knowledge acquisition, we need to turn to the learning science literature. In particular, cognitive load theory, and the example-based learning literature that follows from it help us understand the design of activities explicitly targeting conceptual knowledge acquisition.

Cognitive Load Theory and Example-Based Learning

The acquisition of domain-specific knowledge-structures, or schema (Chi et al.,1981), is the primary function of conceptual learning from a task. Problem-solving practice is not ideal for schema-building since it may involve superfluous production steps that, according to Cognitive Load Theory (Sweller et al., 1998), place a load on the limited cognitive resources of a learner. Instead, the literature on example-based learning shows that worked examples are better for conceptual learning from a task by eliminating these superfluous production steps and allowing learners to focus completely on learning from the problem states showcased (Renkl, 2014).

In prior work, researchers investigated whether worked example-based reflection could work better than problem-solving practice for conceptual learning in the advanced CS context also. It was determined that activity designs that predominantly involved collaborative reflection over worked examples were as good as or better for conceptual learning as well as future programming performance (Sankaranarayanan et al., 2021a, Sankaranarayanan et al., 2021b). Given that this benefit has been revealed, it is no longer necessary to separate programming and collaborative reflection. It may be more effective to combine the two into an "interleaved" design. This kind of "interleaving", has precedent, albeit in the individual context. We turn to cognitive science literature to better understand this.

The self-explanation effect

Cognitive science literature points to the "self-explanation" effect – the benefit gained from students explaining their solution steps during worked example-based reflection (Conati & VanLehn, 1999) for learning from a task, as well as performance on transfer tasks. Later work investigated this effect during problem-solving with the help of cognitive tutors and found similar positive effects -

By engaging in explanation, students acquired better-integrated visual and verbal declarative knowledge and acquired less shallow procedural knowledge. (Aleven & Koedinger, 2002)

The mechanism by which self-explanations lead to learning is described in two parts. First, self-explanation triggers constructive learning processes which require students to actively elaborate on their existing knowledge. Second, this elaboration process could reveal misconceptions that can then be addressed (Conati & Vanlehn, 2000). In the collaborative learning context, we can similarly hypothesize that a prompt for reflection produces elaboration. Only now, this elaboration serves as a scaffold not just for the individual, but for the whole group to have their misconceptions revealed (King, 1997). The resulting discussion, with the right discussion facilitation, can then serve to resolve misconceptions resulting in learning through a process not dissimilar to transactive exchange (Wang et al., 2017). While possible to hypothesize, this particular design of reflection "in the flow" of problem solving, has never been explicitly evaluated in the collaborative learning setting to the best of our knowledge. This contribution of this paper, therefore, is two-fold. First, a design contribution is made with a novel collaborative learning activity for advanced CS contexts. Second, a contribution to CSCL theory is made by investigating the impact of collaborative reflection "in the flow" of problem solving.

Methods

This study tests the hypothesis that interleaving example-based reflection with computer programming presents additional benefits over a phased approach since it increases exposure to reflection opportunities while keeping total time-on-task the same.

Context of the Experiment

This experimental study was conducted in the Summer 2021 semester-long offering of a graduate-level project-based online course titled "Foundations of Computational Data Science". The course serves as an introduction to advanced data science concepts and is offered to graduate students of Carnegie Mellon University.

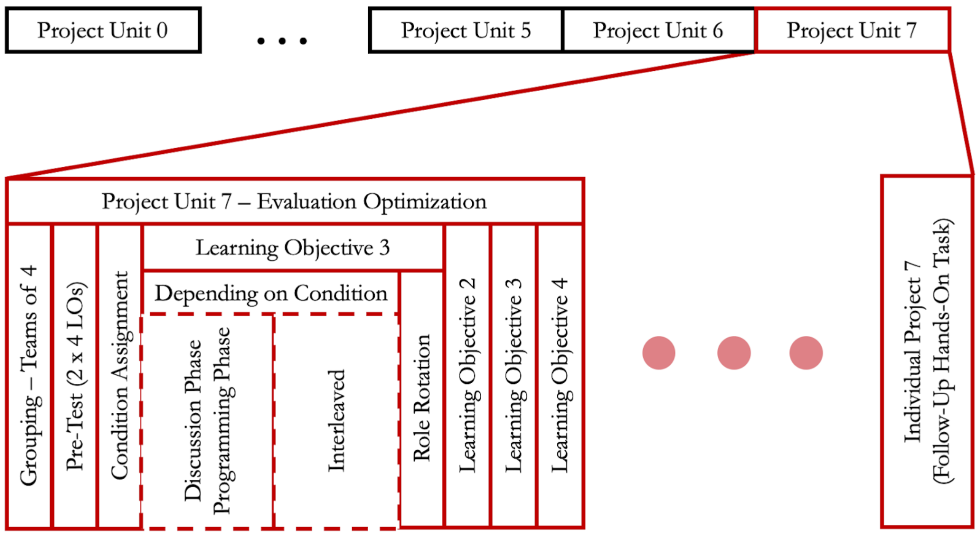
Design of the Collaborative Learning Activity

The course is structured around seven project-based units, each of which culminate in an individual hands-on project with assessment components for the unit. This experiment is within the seventh project unit of the course focused on evaluating and optimizing machine learning models. In preparation for the individual project for the unit, students, in groups of 4, work on the synchronous collaborative programming activity called the Online Programming Exercise (OPE). For the activity, students use a collaborative programming environment called Cloud9 that provides a programming environment and a text-chat interface for communication during programming. A conversational agent, built using the Bazaar framework (Adamson et al., 2014), is also embedded in the chat to provide activity instructions, and facilitation for both reflection and programming.

The activity, overall, lasts 80 minutes and is divided into 4 tasks, each lasting 20 minutes and addressing a granular learning objective (LO). Depending on the condition that each group is assigned to, the task either follows a phased design with separate collaborative reflection and programming phases, or an interleaved design, where both happen together. In either case, a role scaffold is used to assign the four students to four complementary roles – The *Navigator,* the *Driver,* the *Researcher,* and the *Project Manager*. to structure the collaboration. This role scaffold is based on prior work where the Mob Programming paradigm from industry (Wilson, 2015; Zuill & Meadows, 2016) was adapted for use in online instructional contexts (Hilton & Sankaranarayanan, 2019; Sankaranarayanan et al., 2019). The roles rotate after each task such that each team member gets to play the Driver role once during the exercise.

To become accustomed to the role scaffolding and collaborative programming environment, students first read text and watched video content preparing them for the activity in the weeks leading up to it. After that, in groups of 4, they participated in a pilot activity that mimicked the activity the actual study. The content for the pilot was the only difference being kept lightweight in order to retain focus on learning the role definitions and collaborative programming environment. For both the pilot and the study, groups were formed based on students' time-availability that they were polled about earlier in the course. It was ensured that no two students who participated in a pilot activity were in the same group for the experimental study. The position of the experimental study within the course, and the study design are shown in Figure 1 below.

Figure 1
Course Structure and Positioning of Experimental Study: Pre-Test, Post-Test, Activity, and Hands-On Task Alignment



Measurement and Participants

The activity is immediately preceded and succeeded by a pre- and a post-test respectively. The tests have 2 questions corresponding to each learning objective (LO) for a total of 8 questions per test. A score improvement from pre- to post-test serves as a measure of conceptual learning from the task. The individual hands-on programming tasks, due a week later, serve as a measure of conceptual and procedural knowledge, indicating preparation for future programming in this topic. A summary of the location of the pre-test, post-test, and hands-on tasks in the course is also shown in Figure 1. A total of 76 students participated in the exercise and the subsequent project. 10 groups (36 students) were assigned to the phased condition and 11 groups (42 students) were assigned to the interleaved condition. The numbers are not an even 40 and 44 owing to a few no-shows or drops. In a minority of such cases, students participated in groups of 3 with the role scaffolding instructing the student in the third role to play the role of both the Project Manager and the Researcher.

Study Design

The study is conducted using a between-subjects design where students, in groups of 4, are assigned to 2 different conditions - the phased design used in prior work, and the novel interleaved design. Owing to space limitations, one task is used to illustrate the two conditions. Other tasks follow a similar structure but target different learning objectives. Time-on-task for each activity is controlled. Only the distribution of attention to problem solving and reflection differs. Thus, the question is what strategy for allocating attention is the best use of instructional time for producing learning gains.

Phased Design - Example-Based Reflection followed by Programming

Students first participate in the collaborative reflection phase. Here, they are presented with a worked example solution in the form of pseudocode, or an algorithmic outline. The conversational agent prompts comparisons to possible alternative implementations leading to a collaborative reflection about the pros and cons of each approach. Once the ion concludes, and consensus is reached about the approach that the group will use, they enter the programming phase where they are assisted in their implementation through hints provided by the conversational agent. Table 1 illustrates the flow of one task for students in this condition.

Interleaved Design - Reflection “in the flow” of Programming

In this condition, students are presented with two worked example pseudocode solutions. Rather than introducing the alternative implementations as discussion prompts, students are asked to compare and contrast the two presented solutions and choose one that they will implement. This eliminates the need for a separate discussion phase by embedding the conceptual learning in the decision-making process between the two presented solutions. Similar to the phased condition then, implementation assistance is provided through hints. Table 1 illustrates the flow of one task for students in this condition.

Table 1
Task Flow and Example Conversational Agent-Prompts across Conditions

Condition	Phased Design	Interleaved Design
Pseudocodes Presented	1 Pseudocode Presented	2 Pseudocodes Presented
Start of Activity	Start of Collaborative Reflection Phase	Start of Interleaving
Reflection Prompt 1	Led by the Project Manager, take turns to discuss why it's important to zero gradient on each iteration of training in this implementation.	What's the difference between the two pseudocodes?
Reflection Prompt 2	As a follow-up, what will happen if you do not do it? Can you think of an alternate implementation that doesn’t require it?	Why is this difference important?
Phase Transition (If Required)	Start of Programming Phase	No separation of phases necessary
Implementation Hint	For a full training loop example, have the researcher look up https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network	For a full training loop example, have the researcher look up https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network

Analysis and Results

We first verified that learners in both conditions acquired conceptual knowledge from participating in the activities, as measured by pre- to post-test performance. We build a repeated measures ANOVA model with Condition

(Phased vs Interleaved), Time Point (pre- vs post-), and Learning Objective (one of four) as independent variables, and test score as the dependent variable. We include all pairwise and three-way interaction terms as well. The full ANOVA model was significant (*p = .001*) as was the effect of Learning Objective *F(3, 548) = 7.1, p < .0001* and Time Point *F(1, 548) = 5.4, p < .05*. However, neither Condition nor any of the interaction terms were significant, thus there were significant gains between pre- and post-test that were general across both conditions across all learning objectives, though some learning objectives were harder than others. Table 2 shows the raw test scores.

Table 2
Raw test scores on conceptual knowledge for both conditions at pre-test and post-test across all four learning objectives

	LO1: Model-Building Workflow	LO2: CPU vs. GPU Operations	LO3: Model Evaluation	LO4: Applying a Pre-Trained Model
Phased/Pre-test	*.79 (.28 s.d.)*	*.82 (.39 s.d.)*	*.94 (.20 s.d.)*	*.91 (.29 s.d.)*
Phased/Post-test	*.88 (.25 s.d.)*	*.94 (.25 s.d.)*	*.94 (.25 s.d.)*	*.91 (.3 s.d.)*
Interleaved/Pre-test	*.76 (.32 s.d.)*	*.74 (.45 s.d.)*	*.95 (.16 s.d.)*	*.92 (.27 s.d.)*
Interleaved/Post-test	*.88 (.22 s.d.)*	*.84 (.37 s.d.)*	*.97 (.11 s.d.)*	*.97 (.16 s.d.)*

Next, we tested for differential amounts of gain between conditions. We accomplished this by using an ANCOVA model with Condition and Learning Objective as independent variables, pre-test score and time-on-task as covariates, and post-test score as the dependent variable. In this case, the correlation between pre- and post-test score was significant across all learning objectives (*p < .0001*), but there was no significant difference in conceptual learning between the two conditions. Thus, the interleaved design did not show any signs of being detrimental for conceptual learning. In fact, the trend, though not significant, was for more learning to occur in the interleaved design.

Finally, we tested for differential performance on a subsequent hands-on task. Across 9 criteria used to grade the task (ordered in terms of difficulty), students in the Interleaved condition earned a perfect score whereas students in the Phased condition earned 96%. We tested the difference across criteria between conditions using an ANOVA model with Condition and Criterion as independent variables and score as the dependent variable. We also included the interaction term. There was a significant effect of Condition *F(1, 666) = 13.0, p < .0005*, Criterion *F(8, 666) = 2.4 p < .05*, and the interaction *F(8, 666) = 2.3, p < .05*. For the effect of Criterion, the only difference was on the final criterion, for which scores were significantly lower overall. Thus, students performed better on the subsequent programming task, but the advantage was only observed on the final two criteria, which represent the highest difficulty.

Discussion

The results of the study suggest that interleaving collaborative reflection "into the flow" of programming offers advantages over a phased approach. In particular, while we did not find statistically significant advantages in terms of conceptual learning, we did find significant advantages with respect to preparation for future programming. An example conversation, shown below, illustrates that it was indeed the case that students in the interleaved condition are exposed to reflecting based on the presented alternatives for longer durations of the task. They are able to reflect "in the context" of the presented alternatives, going back and forth between implementing and reflection, as necessary.

> (In response to compare and contrast prompt from the conversational agent)
> *Navigator*, noticing the main contrast between the alternatives: "ok well the model has no predict function" (sic)
> *Researcher*: "I'll see what the primer says about predictions"
> *Researcher,* inserts example code from the primer: "You can do something along these lines: <*Example Code*>"
> *Driver,* implements.
> *Navigator*: "it says a list of the two logit scores returned by the prediction output, I don't really know what that means" (sic)
> *Project Manager*, looking at the alternatives again: "can we just try returning logits"
> *Driver*, implements, producing a specific error: "'RobertaTokenizer' object has no attribute 'transform'"
> *Group*, realizes the solution.

Researcher: “We just call it”
PM: “Ohh right”
…
OPE_Bot: “You have passed this test case!”

Like the mechanism for conceptual learning through self-explanations, the reflection prompts require group members to explicitly articulate their mental models. The collaborative context then means that misconceptions, when revealed, are addressed through discussion. Both conditions produce these discussions that result in conceptual learning, albeit in different forms – by reflecting over contrasting solutions in the interleaved design, or by responding to collaborative reflection prompts in the phased design. The interleaved design, however, increases exposure to reflection opportunities by not relegating access to the solutions only to the collaborative reflection phase. The reflection also occurs “in the context” of the solution resulting in beneficial effects similar to those observed from self-explanations prompted during problem solving (Aleven & Koedinger, 2002).

While these results are valuable and suggestive, there are limitations that must be addressed in future work. This study, as well as prior work that used phased designs had inconsistent results in terms of whether the advantages for reflection are related to conceptual learning, preparation for future programming, or both. Results from the placement of the reflection phase before or after the programming phase were also inconsistent resulting in questions remaining about the explanations for the differences in these results (Sankaranarayanan et al., 2021a, Sankaranarayanan et al., 2021b). One idea is that the specific role for reflection in the learning depends on whether the deficits that need to be addressed relate more to the conceptual or procedural aspects of the learning objectives. Another question is whether the differences depend upon the difficulty of the learning objective or the level of preparation of the students prior to the activity. In the current study, one specific order was adopted for a single unit of material and a single student population. Inconsistencies across studies in the prior work suggest that a broader investigation is needed to establish generalizable principles that will enable broad reconsideration of curricula for advanced CS at scale.

Conclusion and Future Work

This paper builds on past work challenging a strongly held belief in CS pedagogy that the most important learning of CS, especially in advanced courses, is through computer programming. Past work in CSCL has offered evidence through phased activities where reflection on worked examples of computer programs either precedes or follows actual computer programming. This past work argues that reflection on worked examples can contribute as much as or more to both conceptual learning and preparation for future programming than actual programming practice. The contribution of this paper is a novel design for interleaving reflection and programming in collaborative programming as well as an empirical evaluation that demonstrates the advantages of the interleaved design over a phased design in terms of preparation for future programming. We determine that interleaving increases the proportion of time available for reflection resulting in performance improvements on future programming. Future work will expand on this finding and address how best to contextualize it within broad reconsideration of curricular practices in advanced CS.

References

Adamson, D., Dyke, G., Jang, H., & Rosé, C. P. (2014). Towards an agile approach to adapting dynamic collaboration support to student needs. International Journal of Artificial Intelligence in Education, 24(1), 92-124.

Aleven, V. A., & Koedinger, K. R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. Cognitive science, 26(2), 147-179.

Chi, M. T., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. Cognitive science, 5(2), 121-152.

Conati, C., & VanLehn, K. (1999, July). Teaching meta-cognitive skills: Implementation and evaluation of a tutoring system to guide self-explanation while learning from examples. In Artificial intelligence in education (pp. 297-304). IOS Press.

Conati, C., & Vanlehn, K. (2000). Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. International Journal of Artificial Intelligence in Education (IJAIED), 11, 389-415.

Garner, S., Haden, P., & Robins, A. (2005, January). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In Proceedings of the 7th Australasian conference on Computing education-Volume 42 (pp. 173-180).

Ericson, B. J., Margulieux, L. E., & Rick, J. (2017, November). Solving parsons problems versus fixing and writing code. In Proceedings of the 17th koli calling international conference on computing education research (pp. 20-29).

Fischer, F., Kollar, I., Stegmann, K., & Wecker, C. (2013). Toward a script theory of guidance in computer-supported collaborative learning. Educational psychologist, 48(1), 56-66.

Hilton, M., & Sankaranarayanan, S. (2019, February). Online mob programming: effective collaborative project-based learning. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (pp. 1283-1283).

Kalyuga, S., Chandler, P., Tuovinen, J., & Sweller, J. (2001). When problem solving is superior to studying worked examples. Journal of educational psychology, 93(3), 579.

Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. Information Processing & Management, 20(1-2), 97-118.

King, A. (1997). ASK to THINK-TEL WHY: A model of transactive peer tutoring for scaffolding higher level complex learning. Educational psychologist, 32(4), 221-235.

Margulieux, L. E., Morrison, B. B., & Decker, A. (2019, July). Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1. In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (pp. 548-554).

Renkl, A. (2014). Toward an instructionally oriented theory of example-based learning. Cognitive science, 38(1), 1-37.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer science education, 13(2), 137-172.

Sankaranarayanan, S., Kandimalla, S. R., Bogart, C., Murray, R. C., Hilton, M., Sakr, M., & Rosé, C. (2021a). Combining Collaborative Reflection based on Worked-Out Examples with Problem-Solving Practice: Designing Collaborative Programming Projects for Learning at Scale. In Proceedings of the Eighth ACM Conference on Learning@ Scale (pp. 255-258).

Sankaranarayanan, S., Kandimalla, S. R., Bogart, C., Murray, R. C., An, H., Hilton, M., Sakr, M., & Rosé, C. (2021b). Comparing Example-Based Collaborative Reflection to Problem-Solving Practice for Learning during Team-Based Software Engineering Projects. In Proceedings of the 14th International Conference on Computer-Supported Collaborative Learning-CSCL 2021. International Society of the Learning Sciences.

Sankaranarayanan, S., Kandimalla, S. R., Hasan, S., An, H., Bogart, C., Murray, R. C., Hilton, M., Sakr, M., & Rose, C. (2020a, June). Creating Opportunities for Transactive Exchange for Learning in Performance-Oriented Team Projects. In The Interdisciplinarity of the Learning Sciences, 14th International Conference of the Learning Sciences (ICLS) (Vol. 3).

Sankaranarayanan, S., Kandimalla, S. R., Hasan, S., An, H., Bogart, C., Murray, R. C., Hilton, M., Sakr, M., & Rosé, C. (2020b, July). Agent-in-the-loop: conversational agent support in service of reflection for learning during collaborative programming. In International Conference on Artificial Intelligence in Education (pp. 273-278). Springer, Cham.

Sankaranarayanan, S., Wang, X., Dashti, C., An, M., Ngoh, C., Hilton, M., ... & Rosé, C. (2019, June). An intelligent-agent facilitated scaffold for fostering reflection in a team-based project course. In International Conference on Artificial Intelligence in Education (pp. 252-256). Springer, Cham.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on software engineering, (5), 595-609.

Soloway, E. (1986). Learning to program= learning to construct mechanisms and explanations. Communications of the ACM, 29(9), 850-858.

Sweller, J., Van Merrienboer, J. J., & Paas, F. G. (1998). Cognitive architecture and instructional design. Educational psychology review, 10(3), 251-296.

Wang, X., Wen, M., & Rose, C. (2017). Contrasting explicit and implicit support for transactive exchange in team oriented project based learning. Philadelphia, PA: International Society of the Learning Sciences.

Wilson, A. (2015, May). Mob programming-what works, what doesn't. In *International Conference on Agile Software Development* (pp. 319-325). Springer, Cham.

Zuill, W., & Meadows, K. (2016). Mob programming: A whole team approach. In *Agile 2014 Conference, Orlando, Florida* (Vol. 3).

Acknowledgment

This work was supported in part by the US National Science Foundation under grants IIS 1822831, and IIS 1917955.