

How to Implement a Peer Instruction-designed CS Principles Course

Beth Simon, University of California, San Diego
Quintin Cutts, University of Glasgow

Abstract

The CS Principles curriculum framework includes explicit learning goals regarding student abilities in communication and collaboration. Computing majors need these skills, but what kinds of activities support the development of these skills, especially in a large lecture course? This paper describes Peer Instruction—a pedagogy developed to support students in developing deep understanding in a lecture environment—and its use in the pilot offering of CS Principles in 2010-11 at the University of California at San Diego.

Peer instruction for deep understanding


One of the exciting things about CS Principles is that it has explicit learning goals regarding student abilities in communication and collaboration. Both of these are skills that we seek to develop in computing majors – but it seems a hard thing to implement in the (large) university lecture. How can we provide students experience in developing these skills? One good approach is to use pair programming in introductory programming courses – which has been shown to have benefits for learning and retention. However, pair programming involves a fairly complex process and may not even be “visible” to the instructor (e.g., if assignments are done outside of a closed lab).

Peer Instruction is a pedagogy developed to support students in developing deep understanding (versus more shallow plug-and-chug abilities) in the standard (and possibly large) lecture environment. Developed by Harvard physicist Eric Mazur after realizing that students could pass his exam but still not “understand” core concepts about forces, the hallmark of peer instruction is that students work in teams talking about challenging questions posed by the instructor.

What does this look like in a CS Principles course? Consider Figure 1—a slide from the second lecture at UCSD. Before class, students are guided in completing preparation for the lecture—that is, we get them exposed to the basic concepts by having them read and follow along in the book developing small Alice programs. In this scenario, we plan that they have basic knowledge of what a “DoTogether” tile should do—and we don’t have a slide “defining” or “teaching” it in the lecture. Instead, we pose this question that engages students in applying their understanding of the “DoTogether” concept in a situation where the answer is far from obvious (note: in Alice the “move up” and “move down” instructions will effectively cancel each other out). Do we really want that, after taking this class,

students can tell us that instructions on a DoTogether tile can cancel each other out if they cause “opposite” behaviors? No. If these students never take another computing course, this wouldn’t possibly have any value to them. However, in the process of discussing this question with peers in a group, students can be engaged in talking about how programs “normally” execute one instruction at a time, they can prompt each other to attend to small details such as the values of parameters, and they can form arguments and provide rationales for why they believe the code behaves a certain way. Additionally, the simple direction to spend class time analyzing and rationalizing about code conveys the key that software (and technology) is something that CAN be understood – that they might have an option other than asking someone else for help or rebooting.

What does this code do?



The image shows a Scratch 'Do Together' block. It contains three instructions in order: 'eskimoGirl say Hello', 'eskimoGirl move up 0.5 meters', and 'eskimoGirl move down 0.5 meters'. Each instruction has a 'more...' dropdown menu.

- A. Makes the eskimo girl say Hello, then jump up and down
- B. Makes the eskimo girl say Hello WHILE jumping up and down
- C. Makes the eskimo girl say Hello
- D. None of the above

Figure 1. A first clicker question designed to prompt explanation and discussion, not just finding the right answer.

Good questions engage students in discussion

There is a specified algorithm by which the Peer Instruction question-asking process occurs (Figure 2). First, students silently consider a question for themselves, and vote using a clicker device. Getting students to commit to an individual answer first not only discourages “free-riding” (or just waiting for your neighbor to do it) and actually prepares the brain to learn by retrieving necessary information to understand the question and try to put together a process for determining a response. At the point (preferably without showing the results of the vote), students can be directed to discuss in their “teams”. For CS Principles specifically, we assigned students to fixed groups of 3 – allowing them to be able to talk to each other in fixed lecture theater seats and enabling them to develop a sense of community and comfort with their team members. Students have told us that they really valued these fixed teams. One student reflected that even though he knows he should attend his (large) lectures, it can be hard to get motivated to do so, when no one will even know if he’s there or not. However, in the CS Principles course he said he came to every lecture – because if he didn’t then his team “would be down one third of their brain” – and he didn’t want to let them down. Post-discussion, students are asked (all) to vote again, perhaps changing their vote based on the discussion. The instructor can ask for volunteers to explain how they thought about the question and then show the results of the graph (the ordering of these steps might vary – depending on the desire to hear explanations of more than just one answer). At this point, the instructor can provide a model of how they

would think about or analyze such a question, or, if necessary clarify any confusion with further explanation or a live-coding demonstration.

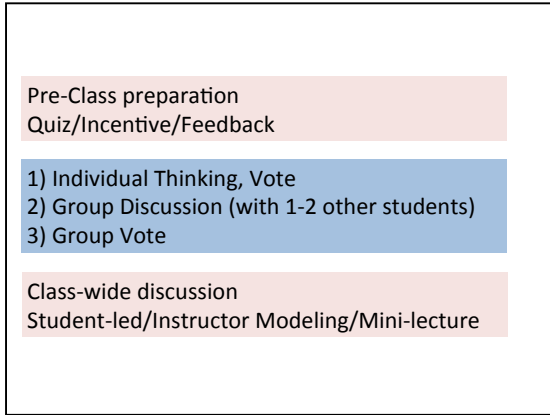


Figure 2: The algorithm for the peer instruction process.

It is critical to develop questions that really engage students in deep and meaningful discussion – rather than in just finding or confirming an answer with their peers. This is where the expertise of the instructor comes into play.

Questions can be devised with a number of goals in mind. One source of good questions is common misconceptions or misunderstandings students have when learning, for example nested loops.

Critical analysis in the context of debugging is a particularly useful skill in CS Principles since the mindset and process can be applied in many technology and software contexts. Figure 4 shows one example of this.

	Outer Loop count	Inner Loop count	Turn parameter
A	4	5	.25
B	4	5	1
C	5	4	.25
D	5	4	1

Goal: Hop in a square, 5 hops on each side

Figure 3. A question to tease out the differences between an instruction in the inner loop and an instruction in the outer loop. Here the fact that the execution of an Alice program is a visual animation supports students in discussing what actually happens – even in the incorrect cases.

A. No, but you could make it better if you include a speed parameter
 B. No, it's perfect as it is
 C. Yes, you have created the wrong parameters
 D. Yes, you have incorrectly used a parameter

Figure 4: This question is presented in the context of talking with someone about some code that they have written. We ask “Do you have any comments for Maria about the code she wrote late last night?” Not only does this question prompt them to consider looking at programs with an eye to “what might be wrong here” but also to double check their understanding of how to use parameters.

Questions can be developed to both check to see that students are thinking about a problem the way we want them to, or that they see the point of a particular concept – perhaps after working a specific problem where they used that concept.

Which of the following is the **best explanation** of what makes a good parameter:

- A. It's something that supports common variation in how the method is done
- B. It's got a meaningful name
- C. It can be either an Object or a number
- D. It helps manage complexity in large programs

Figure 5. A clicker question posed after several questions involving creating parameters and identifying bugs in using parameters.

Even questions that may not have one “right” answer can be used. The question in Figure 6 naturally prompts students to discuss various scenarios where an answer might make sense – and the justification for why.

If we write a method called drive, which would not make sense as a parameter to control how drive occurs?

- A. Destination
- B. How fast
- C. Which car
- D. Car color

Figure 6. A question of design where multiple answers could be defended.

Finally, a major goal of CS Principles is not to create “programmers,” but to use programming to strip away (as much as possible) the complexities inherent in using “real” software applications. At the end of the day, it is through the practice of analysis and justification of answers that Peer Instruction supports that we hope students gain a new approach to dealing with computers and a new confidence in their ability to figure out software and computation in the world around them. This is an outcome that will serve them their entire lives – regardless of career and regardless of the next great app.

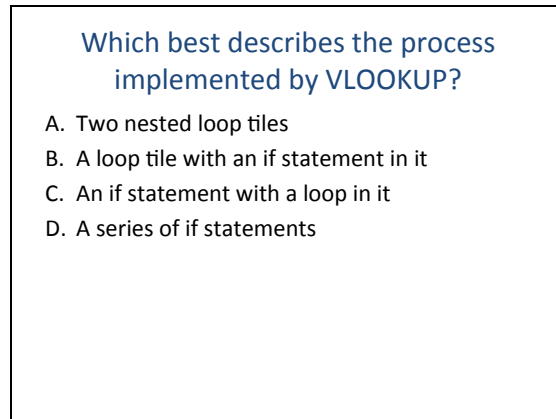


Figure 7. Applying concepts learned in programming to the software and technology around you.

For a more general treatment of peer instruction, see [1]. For further detailed advice on getting started using Peer Instruction in computing courses see [2].

References

1. B. Simon and Q. Cutts, "Peer Instruction: A Teaching Method to Foster Deep Understanding," *Communications of the ACM*, Vol. 55 No. 2, Pages 27-29.
2. <http://cs.ucsd.edu/~bsimon/PI>

Author information

Elizabeth Simon
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
bsimon@cs.ucsd.edu

Quintin Cutts
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland
Quintin.Cutts@glasgow.ac.uk

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education – Computer science education, Curriculum

General Terms: Experimentation, Human Factors, Design

Keywords: Computer science education, pedagogy, CS Principles, peer instruction

© ACM, 2012. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Inroads, {VOL 3, ISS 2, (June 2012)} <http://doi.acm.org/10.1145/2189835.2189858>