Introducing Preservice Science Teachers to Computer Science Concepts and Instruction Using Pseudocode

by Kayla Brauer, Drake University; Jerrid Kruse, Drake University; & David Lauer, Drake University

Abstract

Preservice science teachers are often asked to teach STEM content. While coding is one of the more popular aspects of the technology portion of STEM, many preservice science teachers are not prepared to authentically engage students in this content due to their lack of experience with coding. In an effort to remedy this situation, this article outlines an activity we developed to introduce preservice science teachers to computer science concepts such as pseudocode, looping, algorithms, conditional statements, problem decomposition, and debugging. The activity and discussion also support preservice teachers in developing pedagogical acumen for engaging K-12 students with computer science concepts. Examples of preservice science teachers' work illustrate their engagement and struggles with the ideas and anecdotes provide insight into how the preservice science teachers practiced teaching computer science concepts with 6th grade science students. Explicit connections to the Next Generation Science Standards are made to illustrate how computer science lessons within a STEM course might be used to meet Engineering, Technology, and Application of Science standards within the NGSS.

Introduction

Despite their education typically focusing on science and science teaching, graduates from our science education program are increasingly asked to teach STEM content, including computer science and computational thinking (CS/CT). This article reports on an activity we created to introduce preservice science teachers (PSSTs) to CS/CT so that they might see how to apply sound pedagogical principles if they are asked to teach CS/CT in their future classrooms. Although some may argue CS/CT lies beyond the science classroom, scientists are increasingly using computer science in their scientific endeavors (Wientrop et al., 2016). As scientific data grows increasingly large and complex, analysis requires application of computational thinking (Rubinstein and Chor, 2014). Furthermore, engaging learners in CS/CT creates opportunities to encourage reflection on the nature of engineering (NOE) and the nature of technology (NOT) – concepts argued for inclusion in scientific literacy since the American Academy for the Advancement of Science's *Project 2061* (AAAS, 2000).

While aspects of CS/CT such as problem decomposition, algorithmic thinking, abstraction, and automation are useful problem-solving strategies across disciplines (Yadav, Hong, & Stephenson, 2016), including science, most preservice science teachers (PSSTs) are not

prepared to authentically engage students in this content due to their own lack of knowledge and experience with coding. Programming is a particularly effective way to support students in developing CS/CT ideas (Wing, 2006). However, PSSTs are likely not prepared to design and facilitate coding experiences for their students. Therefore, PSSTs need support in developing both content and pedagogical competency in CS/CT instruction. One way to begin supporting PSSTs in their ability to address CT/CS content in their own future teaching is to engage them in such learning experiences as students. The activity in this article is one way to introduce PSSTs to CT/CS through “unplugging” (Kotsopoulos et al., 2017) by removing the complex technologies often associated with coding. Allowing learners to decode, interpret, and write their own pseudocode without the use of any electronic tools keeps them from getting caught up in programming language specific idiosyncrasies. This unplugged introduction to CT/CS concepts helps to ensure that learners are focused on the logic and concepts rather than syntax issues. This activity is part of a course called “Methods of Engineering and Technological Design” developed to help PSSTs better understand the E and T of STEM education.

The goal of this activity is to engage PSSTs with a common introductory computer science concept, pseudocode. Pseudocode, further defined later, is used by computer programmers to create plain language versions of code in order to work with collaborators who might not understand coding languages (e.g., clients). Through this activity, PSSTs learn about concepts such as conditional statements, problem decomposition, requirements analysis, looping, and debugging. With this foundational knowledge base, PSSTs are then able to explicitly reflect on how teaching pseudocode can serve as platform to engage their future students with the nature of engineering (NOE), the nature of technology (NOT), as well as the NGSS ETS standards.

Brief Background on CS/CT Education

Wing (2006) popularized the term computational thinking (CT) when they argued that CT should be part of students’ curricular experiences. Toward this goal of inclusion, much work has taken place to determine the desired goals or outcomes of computer science and computational thinking (CS/CT). Most broadly, Brennan and Resnick (2012) identified three dimensions of CT: computational concepts, computational practices, and computational perspectives. Building on the notion of computational practices, Weintrop et al. (2016) identified a taxonomy of practices including: data and information, modeling and simulation, computational problem solving, and systems thinking. More recently, Kotsopoulos et al., (2017) paralleled the work of Brennan and Resnick (2012) when noting that CT “involves concepts (e.g., loops, conditions, subroutines) and practices (e.g., abstraction and debugging)” (Kotsopoulos et al., 2017, p.2) that are shared with other disciplines including science, mathematics, and engineering (Kafai & Burke 2013; Lye & Koh 2014).

Research on teaching and learning of CS/CT is still in its infancy (Kotsopoulos et al., 2017). CS/CT teaching typically draws from constructionist frameworks (e.g., Papert & Harel, 1991) in which meaning must be made, but is not achieved only through making. Instead, active mental engagement is required. Yet, like with science, concrete and hands-on practice is important for CS/CT learning (Rubinstein & Chor, 2014). As with all effective teaching, CS/CT should consider students background knowledge and zone of proximal development (ZPD) (Vygotsky, 1978). Students ought to be pushed in their thinking with rigor and logic while avoiding unnecessary technical jargon (Rubinstein & Chor, 2014)

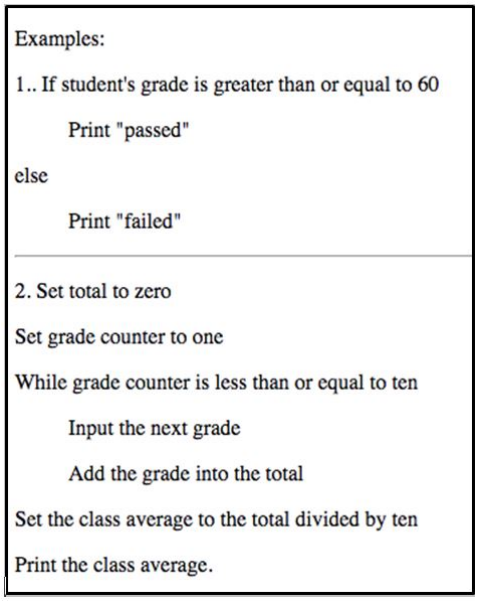
Recognizing the importance of ZPD, Kotsopoulos et al. (2017) conceptualized a pedagogical framework to achieve hands-on practice that included four stages. First, students are introduced to CS/CT through “unplugging” so that concepts can be developed without the barrier of complex computer programming. Second, “tinkering” encourages students to take things apart or modify existing things to understand how they work. Third, students engage in “making” in which they create rather than modify. Finally, “remixing” asks students to apply their knowledge to new situations. Yet, as Papert and Harel (1991) noted, doing/making is not enough to develop CS/CT understandings. To encourage students to make meaning, Rubinstein and Chor (2014) recommend explicit reflection upon computational processes and avoiding the temptation to focus purely on practical instruction.

The instructional activity below uses pseudocode and explicit reflection to help PSSTs wrestle with CS/CT concepts and related CS/CT pedagogical constructs. While the activity described below focuses on unplugging (Kotsopoulus et al., 2017) via the pseudocode activity, other stages (e.g., tinkering, making, and remixing) could be engaged using additional activities, but such activities lie beyond the scope of this article.

Introducing Pseudocode to Preservice Science Teachers

Pseudocode, simply stated, is a detailed, written description of what a computer program or algorithm should do. Pseudocode differs from computer code as it is written in natural language rather than in a programming language. For example, pseudocode is written out as fragmented sentences, whereas computer code often times includes symbols, abbreviations, and other syntax specific to a particular programming language. Pseudocode is written in a way that allows individuals that are not software developers to be able to read and understand the function or purpose tied to a section of programming syntax. Software developers sometimes write pseudocode as a practical exercise intended to help them think through the logic they intend to write into their program’s functions prior to writing the actual code. An example of pseudocode appears in figure 1.

Figure 1 (Click on image to enlarge). Pseudocode example created by Roggio (2014).



Our lesson begins by showing an example of pseudocode (see figure 1). Because the pseudocode is written in plain language, it serves as a more concrete representation through which preservice science teachers (PSSTs) can begin to make sense of computer science concepts. To prompt PSSTs reflection we ask questions such as, “Computer scientists use code to communicate with computers. On the screen is an example of pseudocode, or code written in English rather than a computer programming language. What can we tell from the statements shown in scenario 1?” or “What do you think would happen as a result of these statements?” PSSTs are quick to note that the number 60 is the cutoff point for whether a student passed or failed. However, the second scenario proved to be a bit more challenging. To help students make sense of the words ‘Set’ and ‘While’ we asked, “Why would I want to start the grade counter at one and the total at zero?” and “What purpose is the ‘While’ serving?” or “How do I know when to stop doing the instructions on the two indented lines?” With this guidance, the PSSTs determined that ‘Set’ was a necessary instruction that tells the computer where to start and that the ‘while’ indicates to continue to do something as long as something else is true. In this case, the computer would keep inputting the next grade and adding that grade to the total as long as the grade counter was still less than or equal to ten.

Figure 2 (Click on image to enlarge). Pseudocode example created by Roggio (2014).

```
initialize passes to zero
initialize failures to zero
initialize student to one
while student counter is less than or equal to ten
        input the next exam result
        if the student passed
                add one to passes
        else
                add one to failures
add one to student counter
print the number of passes
print the number of failures
if eight or more students passed
        print "raise tuition"
```

Now that the PSSTs have reasoned through some short sections of pseudocode, we show a more complex set of pseudocode to push their thinking (see figure 2). After allowing some time for students to read through the pseudocode we ask, “What kind of machine do you think this pseudocode might be written for?” When students note that the code seems to be for a digital gradebook we ask, “What aspects of the code helped you make that conclusion?”

We then leverage these pseudocode examples to help the PSSTs engage with additional computer science concepts such as conditional statements. We ask, “What are some words that you see being used consistently throughout each of these blocks of pseudocode?” Some of the PSSTs notice that the ‘while’ statement along with the indented lines were shown in each of the examples as well as words like ‘if’ and ‘else’. We ask the PSSTs to talk in their groups about why the code might use ‘if’ and ‘else’ structures?” After some small group discussion, the PSSTs claim that these structures provide parameters or boundaries for the program or a logic structure for the code to move forward. With this conceptual work done, we name such statements as ‘conditional statements’ and explain that conditional statements give us the ability to check conditions or values of variables and instruct the program to react accordingly. In other words, conditional statements act as checkpoints to ensure that the program is performing in a certain way as long as some other condition is true. For example, in figure 1, the program will print passed if the students’ grade is sixty or greater, otherwise it prints failed. Each time a new grade is fed into the computer program, it has to run against the conditional statement.

We guide the PSSTs to another computer science concept, looping, by pointing to one of the ‘while’ statements and asking, “Why might computer programmers ask the computer to repeat instructions until this condition is met?” The PSSTs quickly explain that having the computer repeat a sequence is a source of efficiency for the programmer. To continue to push students and make connections to the nature of engineering (e.g., the role of trade-offs

in design), we ask, “What problems might programmers run into if they get too hung up on efficiency?” The PSSTs typically note that efficient processes are not always the most productive processes or that efficiency doesn’t matter if the goal is not achieved.

A Pause for Pedagogy

Before asking the PSSTs to apply some of these computer science concepts, we pause the activity to help the PSSTs notice pedagogical decisions by asking, “What have you noticed about how we introduced looping and conditional statements to you?” Because we often have such pedagogical reflections, the PSSTs quickly note that we introduced the academic vocabulary after initial conceptual understanding is developed. We capitalize on this by asking, “How did we support other aspects of academic language development?” Again, because of our students past experiences, they typically note that discussing the conditional statements could be linked to supporting both the syntax and function of language within computer science. To really push students pedagogical thinking we ask, “We often talk about starting lessons as concretely as possible, but this lesson started with pseudocode as words. Why do you think we did this?” This question often requires the PSSTs to discuss in small groups. With some time and perhaps some additional scaffolding questions, the PSSTs recognize that the pseudocode removes the specialized language and is more concrete than showing examples of real code.

Applying Pseudocode to Simple Scenarios

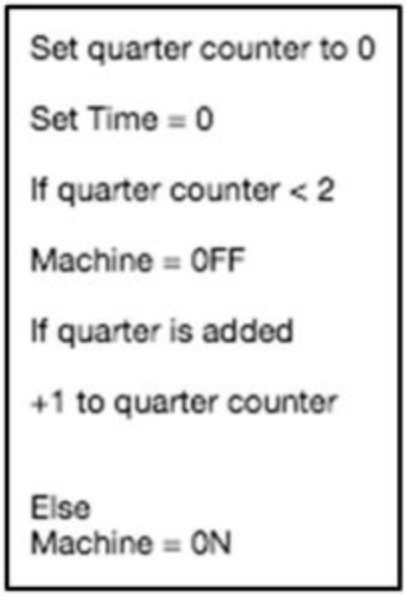
Once the PSSTs can interpret the pseudocode, we move toward writing pseudocode. Prior to showing the video, we tell students that they will be writing their own pseudocode for the machine in the video they are about to watch. We then show a video from Ballen (2007) of an individual playing the arcade game, “Whack-a-Mole”. Immediately after showing the video we ask, “What are some functions that a whack-a-mole machine must be able to perform so that people can play one game?” After giving the PSSTs some time to brainstorm in their small groups, we start to make a list of responses as a class. Some common responses include:

- light up
- add to the score
- update the high score
- randomly move the moles
- know how long to show the moles for
- sound an alarm when time is up

In our experience, the PSSTs did not initially list that the machine has to turn on and off when appropriate. So, before tackling the ideas on the list we tell students that we want to start with a simple example, turning the machine on and off. We ask groups of students to write pseudocode that would turn the machine on and off, noting that two quarters are needed for

this particular game. If a group struggles, we encourage them to consider how if/then structures might help. Most groups were able to write initial code similar to figure 3. If the class seems to struggle, we write “If quarter counter = 1, then…” and ask how they would finish this statement. Once students determine the game would be off, we ask the groups to continue working by considering what might have to precede and come after this line of pseudocode.

Figure 3 (Click on image to enlarge). Pseudocode example.



Now that the PSSTs have written their own pseudocode, we draw their attention to the differences that exist between what they have written and what other groups have created by asking, “I noticed that all of your pseudocode is accomplishing the same goal of turning the machine on and off, but none of your code is exactly the same. Why do you think there were differences in pseudocode from group to group?” At this point, the PSSTs are quick to note that there are multiple ways for a machine to turn on and off or that communicating the function could be done in multiple ways. We build on this thinking by asking, “While there are multiple ways to achieve a goal, what might happen if a software engineer starts to write code before understanding the requirements needed to meet a goal?” The PSSTs fairly easily explain that understanding the details is necessary in order to write the pseudocode otherwise key aspects could be left out. We then label this as “problem decomposition” and “requirements analysis” and ask, “How does this notion of problem decomposition from computer science align to the nature of engineering and the science and engineering practices of the NGSS?” Because of prior conversations about the science and engineering practices within the NGSS, the PSSTs quickly note the connection to “defining problems” as well as “designing solutions”, but this might be a place to introduce these concepts if you’ve not previously done so. We push them to consider the use of models by asking, “In what way is pseudocode an example of modeling?” and “How might pseudocode help computer scientists verify the requirements of the program?”

A Pause for Pedagogy

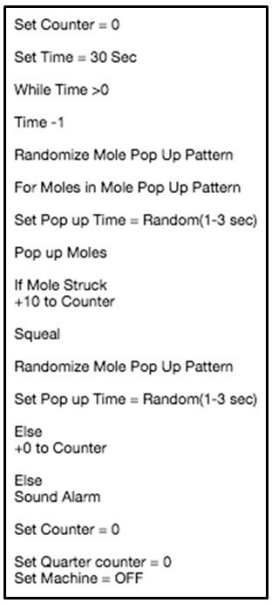
At this point in the activity we pause again to discuss teaching with the students. Sometimes we introduce this conversation with a very open-ended question such as, “What about this activity so far makes it effective?” to see how the PSSTs are conceptualizing the pedagogy behind the activity. More often, we ask a more specific question such as, “Kotsopoulos et al., (2017) would likely call this activity an example of teaching computer science through ‘unplugging’ in which no technical devices or knowledge is needed to learn computer science concepts. What makes this a useful pedagogical approach?” The PSSTs, sometimes with additional scaffolding, provide several rationales for unplugging including connections to the role of language in learning and avoiding technical jargon as well as ZPD issues. For example, one PSST noted that “if we expect younger kids to learn coding by learning javascript, they are likely just going to memorize certain things rather than understand the big ideas of coding”.

We were pleased to see this importance placed on meaning making and noted that this aligns to the thinking of Seymour Papert and others (e.g., Papert and Harel, 1991), but also pushed on this idea by referring to the work of Rubinstein and Chor (2014) and asking, “Why might engaging in some level of actual programming be an important follow up activity?” The PSSTs made connections to science instruction by noting that moving toward actual programming tasks is like moving toward more authentic inquiry experiences in the science classroom.

Writing Pseudocode for Complex Scenarios

Now that the PSSTs have had more experience writing pseudocode, we move to a more complex scenario. We begin by going back to the list generated by the PSSTs earlier in this activity (see section above titled, “Applying Pseudocode to Simple Scenarios”) and direct them to work collaboratively in small groups to write pseudocode for two actions/goals of their choice from the list. For the purpose of this article, we have focused on the task of randomizing the mole movement. We give the PSSTs 10-15 minutes to write pseudocode in small groups. See figure 4 for an example student work. While the PSSTs work, we take opportunities to point out possible use of looping by asking a question like, “How could you incorporate looping to optimize your pseudocode?” Often times, the PSSTs responses to this question were focused on how looping can be used to help them repeat steps or instructions without having to repeatedly write the same lines of code. We then asked, “How will the computer know when to stop the loop?” to help the PSSTs consider how a conditional statement might be necessary to stop the loop.

Figure 4 (Click on image to enlarge). Pseudocode example.



Peer Review as Debugging

Once the PSSTs have written some additional code for the given (or chosen) task, we want them to engage in some peer review to help in debugging their pseudocode. To kick off this portion of the lesson we ask questions like:

- What are some benefits associated with proofreading?
- Why might it be beneficial to proofread your own work *and* have others proofread, too?
- How might looking for errors at the pseudocode level improve efficiency?
- In what way is proofreading like the concept of debugging?

To facilitate the debugging/proofreading, each group designates one member to stay with their groups' pseudocode while the others move to a new group to provide peer review. Depending on time, we may have the PSSTs rotate only once or several times. Once groups have had enough time to work through the debugging exercise, usually about 7-8 minutes, we help the PSSTs make connections to the nature of engineering by asking questions such as:

- Why might software engineers benefit from feedback from people with external perspectives?
- How might debugging and peer review help software engineers optimize their code?

A Pause for Pedagogy and the NGSS

We pause the CS/CT lesson one more time to have a class discussion about pedagogy. We start this discussion by asking, "Notice that we are drawing students' attention to both concepts (e.g., loops, conditional statements) and practices (e.g., debugging) of software engineers. Why do you think we should be asking explicit and reflective question about both?" The PSSTs typically make connections to science teaching noting that in science we

want students to understand both the science content and how scientists work. We then ask, “How might you leverage the practices to make connections to disciplines other than computer engineering?” This could be an opportunity to introduce PSSTs to the science and engineering practices or the mathematical practices, but in our course they have already learned those ideas so our conversation focuses on exploring the connections between the practices of various disciplines.

Understandably, students sometimes wonder why they are learning about computer science in a science education program. To address these concerns, we extend this pedagogy conversation by displaying the NGSS ETS standards:

- A: Defining and Delimiting and Engineering Problem
- B: Developing Possible Solutions
- C: Optimizing the Design Solution

With these displayed, we ask the PSSTs to discuss, “How might this pseudocode activity help students engage with each of these standards. We give the PSSTs time to discuss in their small groups and circulate around the room listening in on conversations and providing prompting as needed. When we come back to our whole-class discussion, the PSSTs typically note that ETS1.A is addressed when they must more clearly define what the whack-a-mole machine is doing before writing the pseudocode. Then, ETS1.B is addressed as students work to write the pseudocode. Finally, ETS1.C is included when peer review and debugging of the code occurs. While we acknowledge the NGSS ETS standards outlined above could be applied to a variety of scenarios or activities, writing pseudocode provides an avenue to engage students with these standards, make connections to CS/CT, and make connections to the natures of engineering and technology.

While we are glad the PSSTs make the connections noted above, we push them a bit further by asking, “Why would students likely not make these connections just by writing pseudocode?” If the PSSTs struggle we ask, “Why are the metacognitive questions we ask about engineering and how engineers work a necessary part of this activity?” With this prompting, and often before it, the PSSTs note that students may complete the task without ever thinking about how they are acting like engineers or how engineers work and that the explicit and reflective questions asked throughout the lesson encourage the students to think beyond just completing the activity.

Embedding the Nature of Engineering and Technology

We hope readers have noticed the many nature of engineering reflective questions throughout the pseudocode activity. Similar to our approach in teaching the ETS standards, we prompt discussions that explicitly focus on the content that we want PSSTs to learn, in

this case, the natures of engineering and technology (e.g., Kruse, Edgerly, Easter, & Wilcox, 2017). Some questions we use throughout the activity to draw PSSTs' attention to the NOE include:

- How was the work that you've done throughout this activity similar to the work of real engineers?
- What are some limitations associated with pseudocode?
- Why was it important to take an iterative approach when developing pseudocode? In what ways do software engineers engage in iterative processes? How does this impact the final software product that they deliver?
- How would the end result of your pseudocode have been different if you had not collaborated with other groups in debugging and making edits? Why do you think software companies encourage collaboration amongst engineers?
- Why is it important for engineers to embrace creativity? How does creativity impact the evolution of software and other technologies?

Beyond discussing how code is developed, we also use the pseudocode and CS/CT discussions as to make connections to NOT ideas by asking questions such as:

- What are some trade-offs associated with using computers to complete a task?
- How do you think the structure of code might influence the way people think?
- How has the use of computers changed our society for better and worse?

Preservice Science Teachers' Plans for Implementation

At the conclusion of this activity, we gave the PSSTs a list of the concepts we had talked about (algorithms, conditional statements, looping, debugging, defining problems, automation, modeling, modularization, and protocols). The PSSTs were each asked to choose two from the list and explain how those concepts could be incorporated into a science lesson that they have taught in the past or plan to teach in the future. We found that the PSSTs tended to focus on four of the CS/CT concepts: (1) the use of conditional statements, (2) using looping to consolidate code, (3) the benefits of modularization, and (4) how scientists practice debugging.

Just over half of the PSSTs mentioned how they would integrate conditional statements into their teaching. One even gave a specific example of a lesson they had taught in which students built paper helicopters (Whirligigs). In order to take this lesson a step further and integrate the concept of conditional statements, the PSST said:

> [As students], If we are designing the flight system of a helicopter, what are some things we would have to get the helicopter to do? Once a list is generated I would ask how could we tell the computer that runs the helicopter how to do these actions? From here I would scaffold to conditional statements and then ask why do computer scientists need to use conditional statements when telling a computer what to do?

Another PSST connected the concept of conditional statements to sound waves with plans to ask students about what a computer needs to know in order to classify a sound wave, and then follow up with an explicit question on implementing conditional statements: "…if a wave has 'x' characteristic then (blank) wave is present. I could also ask them how implementing conditional statements into a computer is an example of automation."

When it came to giving an example of how computer science concepts could be integrated into a science lesson, nearly everyone mentioned that they would have students write or interpret pseudocode. Of course, this outcome was not surprising as the activity focused on pseudocode. However, analyzing the responses helped reaffirm that engaging PSSTs in only one computer science activity is problematic as it will likely result in a limited view of computer science as well as how to integrate computer science concepts into their own teaching.

In addition to their affinity to integrate computer science via pseudocode, multiple PSSTs discussed how they would incorporate looping and ask questions to build students' understanding of how looping can optimize code. However, their responses regarding how they plan to integrate these concepts revealed a more limited understanding of looping and conditional statements. Multiple PSSTs equated "if, then" statements to conditional statements in computer science as evidenced by their end of lesson written responses such as, "if the parachute material allows for greater air catching, then the egg will fall slower towards the ground", "if the light emitted by a planet is 'x' what can you discern?", and "if 125mL flask is used, [then] tare 32.06g from final weight." While these are all "if/then" statements, they do not indicate any connection to their use in computer science to establish loops. This limited understanding further supports the conclusion that one lesson is not sufficient for helping PSSTs fully grasp computer science concepts as well as learn how to engage their own future students with computer science content. Yet, we found this lesson to serve as a good introduction to basic computer science concepts and could be used as a springboard for teaching computer science to PSSTs. However, PSSTs will require additional content learning and experience to gain deeper knowledge of the domain.

Beyond these in-class responses, the PSSTs were also observed trying to integrate CS/CT by the second author during the course's field experience with middle school students. Many of the PSSTs imitated what they experienced in the activity described above and simply swapped the "Whack-a-Mole" example for a new scenario. Others deviated and incorporated code-based games and toys such as Yellies and Lightbot to engage students with computer science content.

Perhaps most interesting was that nearly every PSST was able to plan for and ask explicit and reflective questions to draw students' attention to computer science concepts. In their post activity written responses, one PSST posed questions such as, "Why do computer scientists need to use conditional statements when telling a computer what to do? Why is it helpful for computer scientists to use looping when designing computer programs?" Another

PSST gave an example of an activity in which students would engage in multiple test trials. To start a discussion after this activity, they planned to ask, “How is following these steps over and over during trials like looping in computer science? How could an error in this procedure hinder the results? How is that similar to errors in algorithms?” In reference to modularization, one PSST noted that they would discuss “why modularization is more efficient,” and have each of their students practice modularization by working on small parts of a larger project. The PSST then planned to have students draw on their experiences to inform a discussion on why computer scientists make use of modularization techniques. Another PSST drew on student experiences during their teaching by asking “I noticed that part of the group worked on the roof while another part worked on the walls. How did this benefit your building process?” The, after some discussion the teacher asked, “Why do computer scientists sometimes work on separate parts of code, and then combine them later?”

Final Thoughts

While CS/CT is not often part of science education courses, CS/CT is one way to engage students with the ETS sections of the NGSS. This lesson provides a concrete experience on which the PSSTs can draw when making sense of the ETS standards, coming to grips with the natures of engineering and technology, and creating their own lessons around the ETS standards. While some may argue coding lies beyond the scope of the science classroom, the availability of large data sets and computer modeling makes clear that many scientists engage in CS/CT and the inclusion of coding within science instruction may more accurately model a contemporary nature of science. Unfortunately, many PSSTs do not have the knowledge to engage students in understanding how CS/CT might interact with scientific investigation. Therefore, we created this activity as one way to introduce students to fundamental CS/CT ideas and how they might be more effectively taught to students.

References

American Association for the Advancement of Science. (2000). Project 2061, Science for all Americans.

Ballen, S. (2007, December 26). *Whac-a-mole* [Video file]. Retrieved from https://www.youtube.com/watch?v=K-jaOfIHGko

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Paper presented at the American Educational Research Association. Canada: British Columbia.

Kafai, Y. B., & Burke, Q. (2013). Computer programming goes back to school. Phi Delta Kappan, 95(1), 61.

Kotsopoulos, D., Floyd, L., Khan, S., Namukasa, I. K., Somanath, S., Weber, J., & Yiu, C. (2017). A pedagogical framework for computational thinking. *Digital Experiences in Mathematics Education*, *3*, 154-171.

Kruse, J., Edgerly, H., Easter, J., & Wilcox, J. (2017). Myths about the nature of technology and engineering. *The Science Teacher*, *84*(5), 39.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? Computers in Human Behavior, 41, 51–61.

National Research Council. (2012). *A framework for K-12 science education: Practices, crosscutting concepts, and core ideas*. National Academies Press.

Papert, S., & Harel, I. (1991). Constructionism: Ablex publishing corporation.

Roggio, B. (2014, Dec 27). *Pseudocode examples*. Retrieved from https://www.unf.edu/~broggio/cop2221/2221pseu.htm

Rubinstein, A., & Chor, B. (2014). Computational thinking in life science education. *PLoS computational biology*, *10*(11), e1003897.

Vygotsky, L. S. (1978). Mind in society. Cambridge: Harvard University Press.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*(1), 127-147.

Wing, J. M. (2006). Computational thinking. Communications of the ACM, 49(3), 33–35.

Yadav, A., Hong, H., & Stephenson, C. (2016). Computational thinking for all: pedagogical approaches to embedding 21st century problem solving in K-12 classrooms. *TechTrends*, *60*, 565-568.