Preparing First-Year Engineering Students to Think About Code: A Guided Inquiry Approach

Briana Bettin, Michelle Jarvie-Eggart, *Member, IEEE*, Kelly S. Steelman, and Charles Wallace

***Abstract*—In the wake of the so-called fourth industrial revolution, computer programming has become a foundational competency across engineering disciplines. Yet engineering students often resist the notion that computer programming is a skill relevant to their future profession. Here are presented two activities aimed at supporting the early development of engineering students' attitudes and abilities regarding programming in a first-year engineering course. Both activities offer students insights into the way programs are constructed, which have been identified as a source of confusion that may negatively affect acceptance.**

In the first activity, a structured, language-independent way to approach programming problems through guided questions was introduced, which has previously been used successfully in introductory computer science courses. The team hypothesized that guiding students through a structured reflection on how they construct programs for their class assignments might help reveal an understandable structure to them. Results showed that students in the intervention group scored nearly a full letter grade higher on the unit's final programming assessment than those in the control condition.

The second activity aimed to help students recognize how their experience with MATLAB might help them interpret code in other programming languages. In the intervention group, students were asked to review and provide comments for code written in a variety of programming languages. A qualitative analysis of their reflections examined what skills students reported they used and, specifically, how prior MATLAB experience may have aided their ability to read and comment on the unfamiliar code. Overall, the ability to understand and recognize syntactic constructs was an essential skill in making sense of code written in unfamiliar programming languages. Syntactic constructs, lexical elements, and patterns were all recognized as essential landmarks used by students interpreting code they did not write, especially in new languages. Developing an understanding of the static structure and dynamic flow required of programs was also an essential skill which helped the students.

Together, the results from the first activity and the insights gained from the second activity suggest that guided questions to build skills in reading code may help mitigate confusion about program construction, thereby better preparing engineering students for computing-intensive careers.

***Index Terms*—First-year engineering education, guided inquiry, introductory programming education, MATLAB, quasi-experimental design.**

Manuscript received June 28, 2021; revised November 8, 2021; accepted December 13, 2021. *(Corresponding author: Briana Bettin.)*

This work involved human subjects or animals in its research. The authors confirm that all human/animal subject research procedures and protocols are exempt from review board approval.

Briana Bettin is with the Department of Computer Science and the Department of Cognitive and Learning Sciences, Michigan Technological University, Houghton, MI 49931 USA (e-mail: bcbettin@mtu.edu).

Michelle Jarvie-Eggart is with the Department of Engineering Fundamentals, Michigan Technological University, Houghton, MI 49931 USA (e-mail: mejarvie@mtu.edu).

Kelly S. Steelman is with the Department of Cognitive and Learning Sciences, Michigan Technological University, Houghton, MI 49931 USA (e-mail: steelman@mtu.edu).

Charles Wallace is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA (e-mail: wallace@mtu.edu).

Digital Object Identifier 10.1109/TE.2021.3140051

I. Introduction

THE FUNDAMENTAL shift toward automation of industrial processes, placing computer technology at the heart of engineering disciplines, has been termed a "fourth industrial revolution" [1]. Engineering programs have traditionally acknowledged the importance of computer programming skills for their students and incorporated programming instruction into their common first-year courses, and this need has become all the more imperative in the fourth industrial landscape. First-year engineering instructors, however, are often met with questions from students as to why they need to learn programming when they are not Computer Science (CS) majors. This multidisciplinary research team is investigating methods to reduce barriers to engineering students' acceptance of programming as a necessary skill for their future careers, and to foster their novice programming abilities.

In a recent study of first-year engineering students, previous work found that confusion about "the way programs are constructed" was negatively correlated with confidence, interest, intellectual openness, and the intention to take future programming classes [2]. Here, the term "constructed" can be interpreted in two different ways. A program can be seen as a static artifact, "constructed" in a particular way with its various components contributing to a meaningful whole. But it can also be seen as a dynamic artifact that is "constructed" over time as programmers add to or modify its structure.

Understanding the static structure of computer programs and understanding the dynamic structure of program development are both vital competencies for novice programmers. Thus, bolstering this understanding shows promise not only in encouraging acceptance of programming but also in making students better programmers. This has led to the development of two activities targeted toward "program construction" with the intention of alleviating confusion and thereby increasing acceptance.

Activity 1: Students practice a high-level, structured problem-solving process, used in the introductory CS course at the team's institution, that steers them away from exclusive focus on programming language details. This instruction

invites students to conceptualize and express programming problems in a language-independent way, and it presents a structured process of planning, implementing, and testing programming solutions. This is intended to alleviate student confusion about "the way programs are constructed" in both the static sense (program structure) and the dynamic sense (structuring programs).

Activity 2: Students examine the code written in unfamiliar languages and explore how their nascent programming skills can generalize across programming languages. Students were asked to interpret the code through comments and reflect on their experience doing so. This activity was intended to foster engineering students' perception of beginning programming skills in one language as translatable to other languages, ultimately increasing their sense of the usefulness of learning programming as engineers.

Experimental results in this article indicate that the guided learning of Activity 1 increases student effectiveness in constructing programs. Qualitative data from the results of Activity 2 give insights into how students navigate the structure of a new programming language. Together, these contributions provide guidance to engineering instructors for how to allay confusion, bolster skills, and ultimately cultivate programming as a facet of students' engineering identity.

II. Background

Researchers in CS education have long identified confusion among novice programmers with regard to both the static structure of programs and the dynamic structure of program planning and development. Allan and Kolesar [3] found that "for many students in CS1, it appears that rather than learning the basic concepts of the field, their energies are devoted to learning syntax", and that "rather than learning real problem solving skills, they resort to trial-and-error." Programming faculty tends to emphasize understanding syntax over problem solving within their courses [4]. A lack of focus on problem solving can lead to the development of inaccurate mental models of programs. False mental models can result in incorrect design, more bugs to address during the coding process, and the development of end program bugs that are not correct [4], [5].

In their survey of research on novice programming, Robins *et al.* [6] summarized a number of studies on programming strategies among novice programmers [7]–[9]. They conclude, "instruction should focus not only on the learning of new language features, but also on the combination and use of those features, especially the underlying issue of basic program design", and "the most significant differences between effective and ineffective novices relate to strategies rather than knowledge" [6]. Cutts *et al.* [10] identified "levels of abstraction" in the discourse of experienced programmers, where problem descriptions as "word problems" are broken down into their component algorithmic constructs and concepts (expressed in what the authors call "CS speak"), and then transformed into the literal syntax comprising the program. Clearly, providing students with multiple mental models, uncoupled from programming language syntax, and strategies for solving programming problems is relevant not only to the team's goal of building a programmer identity among engineering students but also a key to making them better programmers.

Researchers focusing on engineering students as novice programmers note similar results. Fangohr identifies "the primary target in teaching computing" to first-year engineering students as "convert[ing] engineering problems into pseudo-code," "a challenging task that requires analytical thinking and creativity" as opposed to "the conversion of this pseudo-code into a program written in one programming language" [11]. As von Lockette observes, engineering students new to programming struggle with the organization of operations in programming, and students who cannot verbally explain the necessary operations of programs generate inoperable or less efficient code [12]. Among electrical and computer engineering students, Anastasiadou and Karakos found that "personal perceptions on the demerit of programming and negative attitudes towards the subject area block the learning process," and conversely that high programming ability leads to acceptance of its usefulness in professional life [13]. This suggests that equipping students with appropriate mental models and strategies may improve the likelihood that first-year engineering students will not only be better programmers but will also more readily identify as programmers.

What works in programming instruction for first-year engineering students? The common choice of MATLAB as the programming language for first-year engineering courses is driven by evidence that its relatively simple syntax contributes to easier adoption among engineering students [14]–[20]. Another promising approach appears to be situating programming activities within a larger engineering problem [21]–[24]. Canfield *et al.* [22] described a programming course for first-year engineering students involving hands-on activities integrating microcontroller hardware; students with this hands-on experience reported greater confidence in their programming skills and a clearer sense of programming's relevance to engineering studies. They report that "the hands-on applications provided a framework on which to build an understanding of the programming constructs." Lumpp *et al.* [24] emphasized that situated programming work in their course helps "dispel the myth that 'I won't need to know any programming' in majors other than CS, electrical engineering or computer engineering."

Where MATLAB is evidenced to have easier adoption among engineering students, is it also able to provide a greater understanding of programming in general? Within CS education, research on transfer suggests it comes in the form of broad ideas and application of prior languages to the new language—even if that application is misinformed. Anderson [25], Wu and Anderson [26], and Harvey and Anderson [27] found that programming skills do not transfer at the time of writing code (WC), but that transfer can be observed in reduced planning time and reading of directions. This suggests that the syntax itself does not transfer in terms of decreased "task completion" time when one is coding, but that having programmed prior transfers some level of

"programmatic planning" skills to approaching problems in a new language. Kurland *et al.* [28] also noted that both students with and without prior programming experience struggled with transfer-related tasks in their study design. They suggested that while novices may be able to write "sophisticated programs," they did not understand beyond "surface code," where experts had abstracted mental models of more general concepts [28].

The literature suggests that engineering students may benefit from interventions helping them move beyond the surface code to build more abstract mental models of programming concepts. The development of these interventions is complicated by the fact that many engineering instructors are trained as engineers and not computer scientists. With little formal training in CS, engineering educators may not always employ best practices from programming education, and they may not have sophisticated mental models of programming concepts to assist engineering students in thinking beyond the use of MATLAB as a tool.

Recognizing this disconnect, this multidisciplinary team identified an opportunity to draw from successful practices in the introductory CS course at the researchers' institution, Introduction to Programming I, CS1121. Best practices used within the CS1121 course include utilizing guided inquiry questions to improve early programming performance. The guided inquiry process constitutes a structured process to problem solving, which has a record of success in CS education, most notably with the Process-Oriented Guided Inquiry Learning (POGIL) framework [29]–[31]. Novice programmers have been shown to naturally think in a series of steps [32] which the POGIL framework further develops.

III. Instructional Context

The two activities were presented in Fall 2020 within a first-year engineering class, Engineering Analysis and Problem Solving (ENG1101), that teaches teaming, ethics, communication, engineering problem solving, and programming in MATLAB. All engineering students within the researchers' university complete this common first-year class. This provides uniform development of essential engineering skills while making transfer between engineering majors easier for students.

ENG1101 is taught in a flipped classroom. Students watch videos and do readings before attending class sessions, where they work actively on assignments in a team environment. Each section is divided into four to five cohorts of approximately 20 students each, with a teaching assistant (TA) dedicated to each cohort. Students work in semester-long teams of four, while the TAs and instructor help teams as needed.

Typical early programming assignments within the class include general engineering problem solving, and often require the application of provided equations. For example, early programming assignments might ask students to write a MATLAB script, which determines the volume and surface area of a cylinder when the radius and height are input by a user. Assignments increase in complexity through the semester, ultimately requiring the use of relational and conditional logic as well as loop structures. During early programming assignments, students are assigned an "algorithm worksheet" (Fig. 1). These worksheets ask students to provide a summary of the problem statement, input/known variables, output/known variables, other variables, assumptions, a written list of steps and/or flowchart, and test cases/conditions.

Summary of Problem:
Determine the volume and surface area of a cylinder.

Sketch:

Input/Known Variables:
r = radius [cm]; h = height [cm]

Output/Unknown Variables:
V = volume [cm3]; SA = surface area [cm2]

Other Variables:
pi = π = 3.14159…

Assumptions:
All units will be given in centimeters; object is a cylinder; and surface area does not include ends

Algorithm (written list of steps and/or flowchart):
1. Get r and h values in cm from user using input function
2. Calculate volume of a cylinder using $V = \pi \cdot r^2 \cdot h$
3. Calculate surface area of a cylinder using $SA = 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2$
4. Report V and SA to user using fprintf function

Test Case:
For r = 3 cm and h = 4 cm;

$$V = \pi \cdot r^2 \cdot h$$
$$V = \pi \cdot (3\ cm)^2 \cdot (4\ cm)$$
$$V = 113.1\ cm^3$$

Fig. 1. Sample algorithmic worksheet from ENG1101.

IV. Overall Study Design

Both activities were delivered in three separate sections of ENG1101. Each section had 100 students, split into five groups. Each group was paired with a near-peer TA. Students within these groups were further divided into teams of three to four students who worked together throughout the semester.

A between-subjects, quasiexperimental design was utilized. Students were not randomly assigned to the intervention and control conditions. Rather, the assignment to condition was done at the group level. In section A, three groups were assigned to the control condition and two were assigned to the intervention condition; in sections B (taught by the second author) and C, two groups were assigned to the control condition and three were assigned to the intervention condition. Due to the COVID-19 epidemic, sections A and B were delivered fully online, while section C used a hiflex hybrid model.

Institutional Review Board (IRB) approval was deemed not required for this study design per the received decision letter. The team obtained a blanket consent for all activities in the study from participants at its onset. Across all three sections, 233 students provided consent to the use of their data for this study and submitted all assignments assessed in this study. Only data from sections A (26 intervention, 39 control) and B (46 intervention, 34 control) are reported here. The initial examination of the data indicated that grades for section C were mostly at or near 100%, and follow-up discussion with the instructor revealed that rubrics were not used in grading.

A. Demographic Context

Study participants were newly enrolled engineering students at a STEM-focused university in the Midwestern United States. Although some may have had previous programming experience, the majority were novice programmers. Roughly

half of the students entering the first-year engineering program have no prior programming experience. Students in the class were predominantly white males, reflecting the undergraduate enrollment at the university, which in 2020 was 88.1% white, 72.6% male, and approximately 11% transfer students. Note that this sample overrepresents white engineering students relative to the distribution found in other US-based engineering programs [33].

V. Activity 1: Guided Questioning

A. Methods

1) Programming Assignments: Within their first week of programming in class, all students completed three programming assignments, two as a group and one as an individual. The first (team) programming assignment asked students to look at the assignment as a team, complete the algorithm worksheet as a team, have their section's TA check their algorithm worksheet, and then write the accompanying program script with their team. Their script was to determine the length of one side of a cube of gold, knowing the specific gravity of gold and requiring the user to enter the mass of the cube.

The second (individual) programming assignment asked students to look at a programming assignment as an individual, complete the algorithm worksheet as an individual, and then write the accompanying script as an individual. Their written script was to require user entry for the Mach number of an aircraft, and given the speed of sound, return the speed of the aircraft to the user in a formatted sentence.

The third programming assignment asked the students to look at a programming assignment as a team, complete and submit the algorithm worksheet as a team, and then write the accompanying script as an individual. Their written script was to determine the altitude at which a satellite should be launched (given the applicable equations and assuming the orbital period of the satellite was equal to that of the EarthâĂŹs rotation) and report that value back to the user.

In the intervention condition only, students were also required to answer a set of guiding questions as they worked through the programming assignments. All three programming assignments presented the following four questions to help examine the assigned problem.

1) What does the user need to provide?
2) What is the program expected to do for the user?
3) What does the program do with the information a user provided?
4) How should the final result be given when the program finishes?

One algorithmic worksheet activity, for example, asked students to explore the following problem.

> *Determine the length of one side of a cube of solid gold, in units of inches. You may assume that the specific gravity of gold is 19.3 and that the user will provide the mass of the cube in units of kilograms. As a test case, you can assume that the user has a 0.4 kilogram cube. — Problem ICA 16-9, "Thinking Like an Engineer," pg. 654 [34]*

An additional three questions were problem specific and the targeted assumptions and conclusions that the programmer should make, test cases, units of calculation, user inputs required by the program, and what information the program must share with the user and how. The three problem-specific questions for the above problem were as follows.

1) What are two assumptions/conclusions you should make given the description "cube of solid gold" that can help in designing your algorithm? (Hint: one relates to your result, the other relates to your calculation of it, and both relate to a different property within that statement.)
2) This problem provided a test case. How should this be used by you for testing? Should this value be added directly to your code?
3) You are asked to make an assumption about the units a user will provide information to you in. What unit is that? Is there a way to make the user aware that this is the expected unit, and if so, what would you do to do this?

Students submitted answers to these seven questions along with their algorithm worksheet.

Prior to submission, students were prompted to additionally reflect on the following set of questions. Students were not required to submit written answers to these questions.

1) Are your steps ordered? Can you number each instruction you provided meaningfully? Make sure this is added to your algorithm.
2) Is every step you provided unambiguous? Did you provide any steps where someone else reading this might interpret the wording you used differently? Did you use any vague words such as "give" or "ask," rather than words that indicate how exactly to "give" or "ask" through the computer?
3) Is your algorithm able to be executed by the computer? Did you make any assumptions about what someone reading the instructions would "just know" that need more specificity for the computer to be able to complete them? Remember, the computer doesnot know anything you didnot tell it to when you are programming!
4) Does your algorithm have a clear, terminable ending where the program can stop? Did you eliminate any "dangling paths" where the algorithm cannot end if certain inputs are provided? No matter what happens during execution, your algorithm should always be able to reach a stopping point, even if it takes a while.

2) Unit Assessment: The unit assessment, totaling 65 points, included a question on engineering unit conversions (5 points), an algorithm worksheet (15 points), and accompanying MATLAB script (45 points). Four versions of the assessment were pseudorandomly distributed to students in each section by assigning one member of each student team a different version of the assessment. This method was selected primarily to discourage cheating among team members testing in an online environment. Each version required the students to write a function (the exact math it performed varied by version) and accompanying script which would require user input of information, call the newly created function, and report results back to the user. A common rubric was used for grading all four versions of the assessment.

B. Procedure

During weeks 2 and 3 of the semester, students completed a module that included prerecorded lecture videos and readings to review before class, check-in quizzes over the preclass materials, and three programming assignments (one individual assignment, sandwiched between two team assignments). The control and intervention conditions differed only in the use of the guided questions. A unit assessment was administered after completion of the module.

Effects of the intervention on performance were assessed by examining the individual programming assignment and the unit assessment. The group programming assignments were not assessed for two reasons: 1) the algorithmic worksheet was completed as a team and 2) the program was graded utilizing MATLAB's Grader tool, which automatically grades students' code and allows unlimited attempts at coding. Students can resubmit their code until competency is achieved. The individual programming assignment, including both the algorithmic worksheet and code, was graded by individual TAs using a common rubric across all sections. Each TA graded programming assignments for only those students in their section. In contrast, within each section of the class, each of the four versions of the unit assessment were graded by a single TA to avoid a confound between graders and condition.

Dependent measures included individual programming assignment score, unit assessment total score, and the three assessment subscores (unit conversion, algorithm worksheet, and MATLAB script).

C. Results

Scores from the individual assignment and the final assessment were submitted to separate two-way ANOVAs, with the condition (control/intervention) and section (A/B) as between-subject factors. Across all variables, the interaction between section and condition was not significant, so only main effects are reported.

1) Individual Programming Assignment: On the individual programming assignment, the main effect of condition was not significant, $F(1, 140) = 3.111$, $—p = 0.08$, and $\eta^2_{—p} = 0.022$; however, there was a trend toward higher scores for students in the intervention condition (M = 9.08, SD = 1.51) than those in the control condition (M = 8.52, SD = 1.94). There was no significant difference in scores between the two sections, $F(1, 140) = 0.00$, $—p = 0.99$, $\eta^2_{—p} < 0.001$.

2) Final Assessment: The final assessment showed a significant main effect of condition, $F\ (1,\ 140) = 11.248$, $—p = 0.001$, $\eta^2_{—p} = 0.074$, with students in the intervention scoring higher (M = 59.23, SD = 8.24) than those in the control condition (M = 54.21, SD = 11.60) (see Fig. 2). Notably, these scores equate to 91% in the intervention condition and 83% in the control condition, nearly a full letter grade difference.

To determine the source of the difference in final assessment score, the team repeated the analysis on each of the three components of the assessment (unit conversion, algorithm worksheet, and programming task). Notably, the effect of condition was significant for the programming task, $F\ (1, 140) = 11.13$, $—p = 0.001$, and $\eta^2_{—p} = 0.07$. However, no differences were observed on unit conversion, $F\ (1, 140) = 1.52$, $—p = 0.22$, and $\eta^2_{—p} = 0.01$ or on the algorithm worksheet, $F(1, 140) = 4.49$, $—p = 0.036$, and $\eta^2_{—p} = 0.03$. Together, these results indicate that performance differences were driven primarily by higher scores on the programming task for those in the intervention condition.

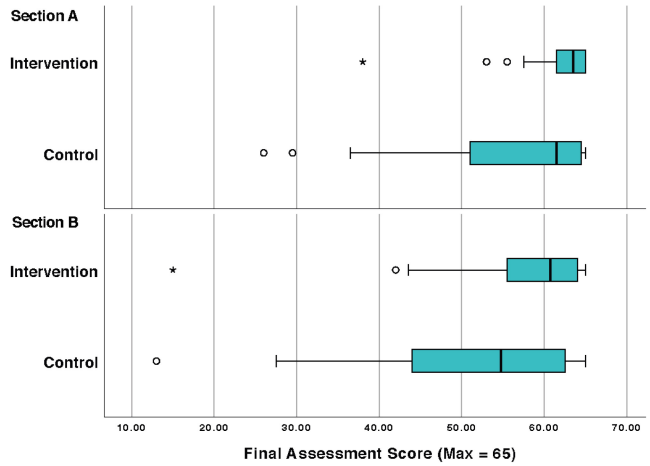

Fig. 2. Box and whisker plot of final assessment scores in the intervention and control conditions. In both sections, the intervention elicited significantly higher scores.

On the final assessment, there was a significant effect of section, $F(1, 140) = 6.04$ $—p = 0.015$, and $\eta^2_{—p} = 0.04$; students in section A scored higher than students in section B on the overall assessment, driven specifically by differences on the algorithm worksheet and programming tasks (as illustrated in Fig. 2, in both sections the intervention elicited significantly higher scores). As TAs completed all of the grading and TAs were nested within the section, it is not clear whether this effect was driven by differences in instruction or differences in grading. As the focus of this study is on the effect of the intervention, the main effect of the section is not discussed further.

To rule out that the observed effects were not driven by differences in performance on the four versions of the final assessment, assessment scores were submitted to a two-way ANOVA, with the test version and condition as between subject factors. Importantly, the effect of condition maintained, $F(1, 136) = 9.06$, $—p = 0.003$, and $\eta^2_{—p} = 0.06$. The effect of test version was nonsignificant, $F(1, 136) = 2.04$, $—p = 0.112$, and $\eta^2_{—p} = 0.04$, as was the interaction between condition and test version, $F(1, 136) = 0.92$, $—p = 0.435$, and $\eta^2_{—p} = 0.02$.

D. Discussion

In summary, Activity 1 produced statistically significant increases in the final assessment score, driven by improvements in the programming task score. Notably, this effect was obtained in two different class sections, with two different instructors, indicating that the effects of the intervention are reproducible. The strength of the effect of the intervention was notable, improving students' assessment scores by nearly

one full letter grade. Thus, guided questioning as an accompaniment to the algorithmic worksheet resulted in better early programming skills than the algorithmic worksheet alone.

An outstanding question concerns the specific aspect(s) of the intervention that is responsible for this performance improvement. Recall that the intervention actually included three parts: a set of four general guiding questions, three problem-specific questions, and a reflection checklist before submitting the assignment. It is, therefore, unclear whether the effect is driven by the combination of all components or if it might obtain with only a subset. Furthermore, as the team did not use an active control condition in this study, individuals in the intervention condition likely spent more time on their assignments than those in the control condition. It is possible that some portion of this effect is driven by time spent on the assignments rather than the nature of the guided questions themselves. Further work is required to better understand the source of the effect. Given, however, that this particular intervention had such a large effect on the final assessment grades, it suggests that regardless of the impact of time on the effect, that this is time well spent.

VI. Activity 2: Transfer of Programming Skills

The goal of Activity 2 was to help students see that their knowledge in one programming language can provide some foundation for understanding concepts within others. Students were given three snippets of the code and were asked to comment on what they thought the code was doing. They were then asked a set of reflective questions concerning their experiences with this task. Qualitative data from the open-ended questions were analyzed using convergent coding methods [35].

A. Methods

Within their last week of class, all students completed one individual assignment where they were asked to read and provide comments on three snippets of the code, each detailing a different task. Afterward, they were asked reflection questions. Although the three tasks were identical for the intervention and control groups, the programming languages differed. Students in the control condition were provided snippets written in MATLAB, which they had just spent the semester learning. Students in the intervention condition were provided snippets of code in other languages: Java, C++, and Python. These languages were chosen after review of the same code snippet results across several languages. The researchers' goal was to choose distinct languages where while keywords would differ, the overall flow of functionality remained similar, in order to see what prior MATLAB skills students used and how. Java and C++ both were used to represent distinct problems that used an array and for each loops. This was considered a "harder" transfer task, as students were required to recognize both new keywords (including static typing), and to navigate the different functionality of a for each loops in Java and C++ which requires separate indexing variable declaration for element access, versus MATLAB's inline for each counter. The for each loop was chosen due to its visual and functional similarity to the MATLAB for loop. Both problems had a perceived similar level of difficulty, but keyword navigation may be different for students between the two languages. The Python problem was considered the "easier" transfer task given nearly identical syntax to MATLAB. Python was chosen for the "function definition" task given somewhat different keywords allowing Python to still look novel and distinct, however, the code result was largely parallel to MATLAB aside from the use of "def" versus "function," and the return statement for results.

The first code sample defined a variable `x` and a vector `y`. It then utilized a `for` loop to compare the value of `x` with each element of `y`. It then calculated the corresponding elements in a vector `z`, using different math based on whether an individual element of `y` was greater than `x`. This code was provided in either MATLAB or Java.

The second code sample defined a variable `--p` and a vector `A`. It requested user input for the value of `Q`. Within a `for` loop, it determined if each element of `A` was less than `--p`. If it was, the value `Q` was added to the value of that element. This code was provided in either MATLAB or C++.

The third code sample utilized a function with two input variables `C` and `D` and one output vector `G`, which was calculated using the input variables. This code was provided in either MATLAB or Python.

For each code example, students were asked to add comments to the code, explaining what each line does. For both conditions, students were asked two reflection questions.

Q1: What skills did you learn in this class that helped you read and comment code?

Q2: What extent do you think learning MATLAB code is helpful in making sense of programs written in other languages?

Students in the intervention were additionally asked the following.

Q3: What similarities did you notice across these languages?

B. Analysis

A grounded theory methodology was used for a qualitative analysis of the reflective question responses. Within the grounded theory, concepts constructing the theory are derived from the data itself, and not preconceived by the researchers [36].

All four authors read through students' responses and collaboratively generated provisional codes. Using this initial code book, a comparative analysis was performed using convergent coding methods where two researchers (Authors 1 and 2) independently coded results and met to discuss and resolve coding differences. With each coding cycle, the researchers updated the code book and definitions until interpretive convergence was met [35]. Simultaneous coding methods were applied [35], where more than one code can be applied to the same text passage, when appropriate.

C. Positionality Statement

The research team included faculty from CS, Cognitive and Learning Sciences, and Engineering Fundamentals. The convergent coding was performed by the faculty member from

Codes	Total	Control	Intervention
SC: Syntactic constructs	73	37	36
Landmarking:	46	5	41
RC: Recognizing syntactic constructs	29	2	27
RL: Recognizing lexical elements	11	1	10
RP: Recognizing Patterns	6	2	4
US: Understanding Static Program Structure	34	24	10
UD: Understanding Dynamic Program Flow	24	20	4
MV: Misinformed vocabulary	19	9	10
WC: How to write code	18	10	8
Prior Experience:	14	7	7
PE: Prior Experience	7	3	4
PE-: Lack of prior experience	7	4	3
LN: Learned nothing new	7	3	4
PS: Problem solving skills	9	1	8

Fig. 3. Activity Q1: What skills did you learn in this class that helped you read and comment code?

Codes	Total	Control	Intervention
Helpfulness:	147	70	77
HE: Help	89	39	50
DH: Doesn't help	8	5	3
NE: Neutral	15	10	5
NA: Did not address	35	16	19
Landmarking:	59	26	33
RC: Recognizing syntactic constructs	23	14	9
RL: Recognizing LE	17	3	14
RL-: Differences in LE confusing	13	8	5
RP: Recognizing Patterns	6	1	5
US: Understanding Static Program Structure	31	20	11
Prior Experience:	28	18	10
PE: Prior Experience	14	11	3
PE-: Lack of prior experience	14	7	7
MV: Misinformed vocabulary	28	8	20
EL: Easier to Learn	24	18	6
MA: MATLAB Descriptions	20	5	15
MA-: MATLAB Differences cased confusion	10	5	5
UD: Understanding Dynamic Program Flow	14	9	5
WC: How to write code	13	8	5
GI: Guess/Intuition	15	5	10
MA-: Matlab Difficulties	10	5	5
CL: Class	8	3	5
PS: Problem solving skills	8	6	2
SL: Simultaneous Learning Experience	5	5	0

Fig. 4. Activity Q2: To what extent do you think learning MATLAB code is helpful in making sense of programs written in other languages?

CS, who routinely teaches introductory programming classes, and the faculty member from Engineering Fundamentals, in whose class the activities from this article took place. Coding activities were performed after grades were calculated for the semester.

D. Results

Common conceptual themes identified within the student reflections were identified, with major themes described here. Fig. 3 shows the coding analysis results for question 1, Fig. 4 shows the results of question 2, and Fig. 5 shows the results of question 3. Note that question 3 was only asked within the intervention condition and, thus, a breakdown of control versus intervention is not provided. Also, note that these results are qualitative in nature and can not be interpreted to judge differences in effectiveness between the control and intervention group.

1) Syntactic Constructs: When students were asked about which skills learned in the class helped them to read and comment code, the most prevalent theme mentioned was syntactic

Codes	Total	Control	Intervention
Syntactic Constructs:	70	-	70
VA: Variables	46	-	46
LP: Loops	45	-	45
IF: If statements & conditions	32	-	32
VE: Vectors, arrays, & matrices	31	-	31
IO: Input/Output	6	-	6
FN: Functions	6	-	6
SP: Suppressing	5	-	5
Landmarking:	67	-	67
RL: Recognizing lexical elements (LE)	28	-	28
RC: Recognizing syntactic constructs	25	-	25
RP: Recognizing Patterns	14	-	14
DF: Differences across languages	13	-	13
MV: Misinformed vocabulary	10	-	10

Fig. 5. Activity Q3: What similarities did you notice across these languages?

constructs (SC). This code, SC, was applied to comments about how to create specific coding elements or what the elements do, such as plotting, creating vectors, functions, variables, loops, conditionals, and MATLAB syntax. The SC theme was identified in 73 (49.7%) of the responses to Q1. One student responded to the question by explaining "I learned how to define variables, vectors, and matrices. I learned how to create a for loop and nest if then statements within the for loop." indicating that learning how to create these specific SCs assisted them in being able to interpret code written by others. Interestingly, a similar number of students in the control and intervention conditions (37 student responses within the control condition and 36 in the intervention) mentioned their knowledge of how to create and identify SCs as a skill that aided them in this exercise. Whether students were working in a language familiar to them or not, beginning programming knowledge of how to create SCs aided them in being able to make sense of the code they had not written.

Not surprisingly, SCs were also the most commonly identified theme within Q3, which asked what similarities were noticed across languages. Q3 was only answered by 77 students in the intervention, for which 70 (90.9%) responded by listing at least one SC including, in order of decreasing prevalence: variables, loops, conditionals, vectors, input/output, functions, and suppression (the use of semicolons in MATLAB to suppress Command Window display), indicating that these students were recognizing SCs in other languages.

2) Landmarking: Landmarking emerged as one of the most common themes for all three questions. Landmarking refers to students *referencing* SCs (RC), lexical elements (RL), or recognizing macrolevel patterns (RP) to make sense of code they did not write. These familiar features can be thought of as navigation aids that help students make their way through an unfamiliar program.

In response to Q1, the landmarking theme was mentioned by 46 (31.3%) of respondents, 41 of whom were in the intervention. For those students trying to make sense of code written in an unfamiliar programming language, landmarking was a specially useful skill. Recognition of SCs (RC) was the most commonly mentioned landmarking skill, composing 63.0% of the responses in this theme for Q1. One student explains how recognition of SCs helped them to read and comment the code, "I recognized some stuff like setting variables and creating

matrix's [sic]." Recognition of lexical elements (RL) was the second-most common landmarking skill, with 23.9% of the responses in this theme for Q1. In the words of one student, "From this class, I knew what different syntax meant, and I assume that it is pretty similar across all languages. For example, I know that at least for MATLAB, square brackets usually mean it will be a vector or that for, else, and elseif are pretty similar across the languages." Recognizing patterns (RP) composed 13.0% of the responses in the landmarking theme for Q1. One student responded to question one by explaining the skill they learned in class was "trying to decipher the syntax and looking for patterns that indicate certain commands."

When asked the second question "To what extent do you think learning MATLAB code is helpful in making sense of programs written in other languages," the landmarking theme emerged as the most common, mentioned by 59 (40.1%) respondents. However, there was a more equal split between the intervention (33 respondents) and the control (26), than in the response to Q1. It seems then that students in both the intervention and the control were likely to see the benefit of landmarking skills learned while learning MATLAB. One interesting difference in the results for Q2 was the emergence of negative comments around recognizing lexical elements. 13 respondents (eight in the control and five in the intervention) specifically mentioned believing the differences in lexical elements caused confusion when using MATLAB to make sense of other languages. As those in the control did not actually read code in a language other than MATLAB, note that their responses are based on speculation or previous experience outside this task. In fact, the most common code for the landmarking theme within the intervention group was that of lexical elements, with students indicating that lexical elements were helpful in interpreting unfamiliar languages.

Of the 77 respondents to Q3 in the intervention group, 67 (87.0%) indicated they utilized landmarking in some way to make sense of code written in unfamiliar languages. Lexical elements were identified by 28 (36.4%) of respondents; SCs by 25 (32.5%); and patterns by 14 (18.2%).

3) Static Structure: Understanding Static Program Structure (US) emerged as another major theme, the third-most common in Q1 and second-most common in Q2. Comments given the US code pertained to understanding order, code logic, or the structure of the language itself. Students referred to reading the code as a book, or explaining it to others. With regards to Q1, US was useful for helping students read and comment code, as noted by 34 (23.1%) of respondents, 24 of which were in the control condition. In the words of one student "I have learned to read lines of code in parts and figure out what each piece does." With regard to Q2, US was mentioned by 31 (21.1%) of respondents, 20 of whom were in the control condition. For both Q1 and Q2, approximately twice as many respondents in the control mentioned US than in the intervention condition. Those in the control were reading examples of the code in MATLAB, which they had just learned, and may have been more easily able to understand the examples they read, leading to them to be more likely believe this to be an important skill. The team sees a similar pattern with the importance of understanding the dynamic program flow.

4) Dynamic Flow: Understanding Dynamic Program Flow (UD) was the third-most commonly mentioned theme in response to Q1, mentioned by 24 (16.3%) of respondents, 20 of which were in the control condition. It was also mentioned in response to Q2 by 14 (9.5%) respondents, nine of which were in the control. Responses coded with this theme pertained to the way code flows, making flow charts or trace tables, and following through loops. Like US, those students viewing code examples in a familiar language mentioned the value of UD more often than those viewing examples of code in unfamiliar languages.

WC, Problem Solving Skills (PS), and Intuition (GI) were also themes that emerged in Q1 and/or Q2, however, each was named by fewer than 15% of students and is not discussed further for brevity.

E. Discussion

Activity 2 provided insights on how novice programmers read and navigate new programming languages. Understanding SCs and landmarking both emerged as essential skills. Notably the ability to recognize these SCs is necessary for effective landmarking. Other landmarking skills include recognizing lexical elements and patterns. One interesting finding was that lexical elements that were similar but not identical were perceived as helpful by some but confusing by others. This is consistent with observations from the extant literature that when learning a new programming language, concepts from previous languages may cause both facilitation and interference, where prior knowledge can both help and hinder depending on the notions one attempts to transfer [37], [38]. Students may benefit from discussions of unique and similar lexical elements across some languages. As students increasingly enter the college with some background programming experience, overtly pointing out these contrasts within the classroom, or designing student reflections about the differences in lexical elements of languages they may know, may be useful strategies for resolving confusion around differences in lexical elements.

Understanding the static structure and flow of programs also emerged as essential skills in making sense of code in unfamiliar languages, as well as helping them understand the code written by others in a language they know. The researchers' past work revealed students who do not understand how programs are constructed feel less confident about programming and have less intention to take future programming classes [39]. Results here point to aspects of the static structure that might be targeted to help students better understand program construction. Also, dynamic flow was much more frequently noted by those interpreting the code in a familiar language than those with an unfamiliar language (83.3% of the UD coded responses to Q1), suggesting that the dynamic structure may be more difficult to transfer, or may not be a skill students leverage readily when navigating a new language.

Activity 2 yielded rich qualitative data to help guide future work. One limitation is that the team did not assess how accurate students were in commenting the code, so the results here reflect only students' subjective account of the factors that they found helpful, with no basis for the extent to which they were actually successful at the task. Future work should use mixed-method approaches to examine this further.

VII. Conclusion

Activity 1 demonstrated that guided questioning during preprogramming algorithm development improves programming skills. Due to its success, it has been implemented across the first-year engineering program. The algorithmic worksheets have now been revised and all include reflective questions. Activity 2 shows how novice programmers use SCs, which are essential for landmarking while navigating new programming languages, along with lexical elements and macrolevel pattern recognition. US and UD may also be essential to making sense of new languages. Together, these activities show that change can be effected in engineering students' ability to make sense of programs, both through the development of programming skills and exposure to examples of code written in other languages. This ability to make sense of programs bolsters their ability to program, and in turn, may lead to greater acceptance of programming as part of engineering careers. Instructors are preparing engineers to enter workplaces where they are increasingly likely to encounter code within their careers across a myriad of languages. It is essential that engineers of the future be capable and confident in making sense of other's code and in learning new programming languages.

The combined success of the first activity and insights gained from the second activity shine a light in the directions of future research for this team. The guided questions of Activity 1 modeled for students the types of questions that experienced programmers typically ask when writing programs, while the analysis of data from Activity 2 revealed what students see when they are reading code in new programming languages. The types of questions one asks when reading versus WC are different, and understanding both processes is essential to success as a programmer. As engineers will need to interact with other programmers in their future careers, working to develop skills in reading code may be specially fruitful to their future careers. Modeling expertise in reading code by using guided questions like those in Activity 1 is a promising area for future work.

Beyond these activities, the team has found significant success in a multidisciplinary approach to this work. Researchers from CS education, engineering education, and cognitive and learning sciences collaborated to address the problem of engineering student programming acceptance and to design curriculum aligned with best practices while tailored to these students and their context. The team leveraged best practices for teaching programming skills while considering "what works" for engineers and their perceptions and goals. Together, the team approached the design problem through an understanding that engineers are required to use programming skills in their career. The research team's blend of perspectives and knowledge areas prompted innovation and the discovery of new research questions time and time again. From these experiences, the research team recommends leveraging multidisciplinary teams in creating transdisciplinary education practices for the modern engineer.

Appendix

A. Blank Algorithmic Worksheet Used for Activity 1

Summary of Problem:	
Sketch:	**Input/Known Variables:**
	Output/Unknown Variables:
	Other Variables:
	Assumptions:
Algorithm (written list of steps and/or flowchart):	
Test Case:	

Answer the following questions in writing and make any changes needed to your worksheet above:

1. What does the user need to provide?
2. What is the program expected to do for the user?
3. What does the program do with the information a user provided?
4. How should the final result be given when the program finishes?

Before submitting your Algorithm, take time to consider the following questions. Make any changes needed in your algorithm worksheet:

1. Are your steps ordered? Can you number each instruction you provided meaningfully? Make sure this is added to your algorithm.
2. Is every step you provided unambiguous? Did you provide any steps where someone else reading this might interpret the wording you used differently? Did you use any vague words such as "give" or "ask", rather than words that indicate how exactly to "give" or "ask" through the computer?
3. Is your algorithm able to be executed by the computer? Did you make any assumptions about what someone reading the instructions would "just know" that need more specificity for the computer to be able to complete them? Remember, the computer doesn't know anything you didn't tell it to when you are programming!
4. Does your algorithm have a clear, terminable ending where the program can stop? Did you eliminate any "dangling paths" where the algorithm cannot end if certain inputs are provided?
5. No matter what happens during execution, your algorithm should always be able to reach a stopping point, even if it takes a while.

B. Rubric for Unit 2 Assessment Used in Activity 1

Algorithm		
2 pts each for the following: Problem Statement, known/input, unknown/output, and test case; 1 pt for each of the following: assumptions and sketch; 5pts for their algorithm/pseudocode/flowchart (values from user, initialize variables, perform calculation in function, perform desired calculation, report values to user)		
Code executes without errors?		
5 pts: yes	0 pts: No	
Code produces correct answer?		
5 pts: yes	0 pts: No	
Script: use of input function		
5 pts: Correct use of input function, test case parameters are used, and prompts to user are useful	2.5 pts: Evidence of input, but incorrect (either did not use provided test parameters, or incorrect syntax)	0 pts: No evidence of input function
Script: Function call		
5 pts: Function call correct (includes all needed input and output variables)	2.5 pts: Evidence of function call present, but incorrect	0 pts: No evidence of function call
Script/function: Calculation of mass		
5 pts: Correctly calculated	2.5 pts: Evidence that calculation was attempted but incorrect	0 pts: No evidence of calculation
Script: use of fprintf		
5 pts: Correct use of fprintf, answer includes units	3 pts: Evidence that fprintf was attempted but not correct	0 pts: No evidence of fprintf present
Script: Comments (header and throughout)		
3 pts: Useful header comments and comments throughout code are present	1.5 pts: Missing either header comments or comments throughout code explaining steps	0 pts: No evidence of comments in script
Function: Definition Line		
5 pts: Correct	2.5 pts: Evidence to create function definition line is present	0 pts: No evidence of function definition line
Function: Calculation		
5 pts: Correctly used equation	2.5 pts: Evidence for calculation is present but incorrect	0 pts: No evidence of function definition line
Function: Comments (header only)		
2 pts: Useful header comments are present	1 pts: Evidence of comments, but insufficient	0 pts: No evidence of comments

ACKNOWLEDGMENT

The research team thanks the Michigan Technological University IDEA Hub program for funding to assist the implementation and analysis of these results. The ENG1101 instructional team, including undergraduate LEarning with Academic Partners (LEAP) leaders, the undergraduate TAs, are also thanked immensely for incorporating the instructional interventions. Dr. Amber Kemppainen is thanked for assistance in administering the study in her first-year engineering classroom. Finally, the team's undergraduate researcher, Jesse Jacobusse, is thanked for her assistance in helping with project preparations and maintenance.

REFERENCES

[1] K. Schwab, *The Fourth Industrial Revolution*. New York, NY, USA: Crown Publ. Group, 2017.

[2] B. Bettin, M. Jarvie-Eggart, K. Steelman, and C. Wallace, "Infusing computing identity into introductory engineering instruction," in *Proc. IEEE Front. Educ. Conf. (Section T8-FY2-5)*, 2020, pp. 1–5.

[3] V. H. Allan and M. V. Kolesar, "Teaching computer science: A problem solving approach that works," *ACM SIGCUE Outlook*, vol. 25, nos. 1–2, pp. 2–10, 1997.

[4] C. S. Cheah, "Factors contributing to the difficulties in teaching and learning of computer programming: A literature review," *Contemp. Educ. Technol.*, vol. 12, no. 2, 2020, Art. no. ep272.

[5] L. Winslow, "Programming pedagogy—A psychological overview," *SIGCSE Bull.*, vol. 28, pp. 17–22, Sep. 1996.

[6] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Comput. Sci. Educ.*, vol. 13, no. 2, pp. 137–172, 2003.

[7] J. Spohrer and E. Soloway, "Novice mistakes: Are the folk wisdoms correct?" in *Studying the Novice Programmer*, E. Soloway and J. Spohrer, Eds. Hillsdale, NJ, USA: Lawrence Erlbaum, 1989, pp. 401–416.

[8] D. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in novice programmers," in *Studying the Novice Programmer*, E. Soloway and J. Spohrer, Eds. Hillsdale, NJ, USA: Lawrence Erlbaum, 1989, pp. 261–279.

[9] S. Davies, "Models and theories of programming strategy," *Int. J. Man-Mach. Stud.*, vol. 39, no. 2, pp. 237–267, 1993.

[10] Q. Cutts, S. Esper, M. Fecho, S. R. Foster, and B. Simon, "The abstraction transition taxonomy: Developing desired learning outcomes through the lens of situated cognition," in *Proc. 9th Annu. Int. Conf. Int. Comput. Educ. Res.*, 2012, pp. 63–70. Accessed: Dec. 16, 2021. [Online]. Available: http://doi.acm.org/10.1145/2361276.2361290

[11] H. Fangohr, "A comparison of C, MATLAB, and Python as teaching languages in engineering," in *Computational Science (ICCS)* (Lecture Notes in Computer Science 3039), M. Bubak, G. D. van Albada, P. M. Sloot, and J. Dongarra, Eds. Heidelberg, Germany: Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-3-540-25944-2_157

[12] P. von Lockette, "Algorithmic thinking and MATLAB in computational materials science," in *Proc. ASEE Annu. Conf.*, 2006, pp. 1–13.

[13] S. Anastasiadou and A. Karakos, "The beliefs of electrical and computer engineering students regarding computer programming," *Int. J. Technol. Knowl. Soc.*, vol. 7, no. 1, pp. 37–51, 2011.

[14] E. Yaz and A. Azemi, "Utilizing MATLAB in two graduate electrical engineering courses," in *Proc. Front. Educ. Conf.*, 1995, pp. 1–5.

[15] J. Vondrich and E. Thondel, "The application of MATLAB in engineering education," in *Proc. Int. Conf. Simulat. Multimedia Eng. Educ.*, 2003, pp. 19–23.

[16] M. S. Habib, "Enhancing mechanical engineering deep learning approach by integrating MATLAB/simulink," *Int. J. Eng. Educ.*, vol. 21, no. 5, pp. 906–914, 2005.

[17] M. A. Wirth and P. Kovesi, "MATLAB as an introductory programming language," *Comput. Appl. Eng. Educ.*, vol. 14, no. 1, pp. 20–30, 2006.

[18] K. F. Larsen, N. A. Hossain, and M. W. Weiser, "Teaching an undergraduate introductory MATLAB course: Successful implementation for student learning," in *Proc. ASEE Annu. Conf. Expo.*, 2016, pp. T3B-1–T3B-23.

[19] R. de Guzman, J. C. Vaccaro, A. H. Pesch, and K. C. Craig, "Freshman engineering problem solving with MATLAB for all disciplines," in *Proc. ASEE Annu. Conf. Expo.*, 2016, p. 16. [Online]. Available: https://peer.asee.org/collections/44

[20] D. Belfadel, M. Zabinski, and I. Macwan, "Introduction to MATLAB programming in fundamentals of engineering course," in *Proc. ASEE Annu. Conf. Expo.*, 2021, p. 11. [Online]. Available: https://peer.asee.org/collections/109

[21] J. Bowen, "Motivating civil engineering students to learn computer programming with a structural design project," in *Proc. ASEE Annu. Conf.*, 2004, pp. 1–11.

[22] S. Canfield, S. Ghafoor, and M. Abdelrahman, "Enhancing the programming experience for first-year engineering students through hands-on integrated computer experiences," *J. STEM Educ.*, vol. 13, no. 4, pp. 43–54, 2012.

[23] K. M. Kcskemety, Z. Dix, and B. Kott, "Examining software design projects in a first-year engineering course: The impact of the project type on programming complexity and programming," in *Proc. IEEE Front. Educ. Conf.*, 2018, pp. 1–5.

[24] J. K. Lumpp *et al.*, "Instrumentation and measurement in a first-year engineering program," *IEEE Instrum. Meas. Mag.*, vol. 21, no. 3, pp. 20–35, Jun. 2018.

[25] J. R. Anderson, "Transfer of skills among programming languages," Dept. Psychol., Carnegie Mellon Univ., Pittsburgh, PA, USA, Rep. ARI Research Note 95-40, 1995.

[26] Q. Wu and J. R. Anderson, "Knowledge transfer among programming languages," in *Proc. 5th Annu. Workshop Psychol. Program. Interest Group*, Dec. 1992, p. 6. Accessed: Dec. 16, 2021. [Online]. Available: http://ppig.org/library/paper/knowledge-transfer-among-programming-languages

[27] L. Harvey and J. R. Anderson, "Transfer of declarative knowledge in complex information-processing domains," *Human Comput. Interact.*, vol. 11, no. 1, pp. 69–96, 1996. Accessed: Dec. 16, 2021. [Online]. Available: https://doi.org/10.1207/s15327051hci1101_3

[28] D. M. Kurland, R. D. Pea, C. Clement, and R. Mawby, "A study of the development of programming ability and thinking skills in high school students," *J. Educ. Comput. Res.*, vol. 2, no. 4, pp. 429–458, 1986. Accessed: Dec. 16, 2021. [Online]. Available: https://doi.org/10.2190/BKML-B1QV-KDN4-8ULH

[29] R. S. Moog and J. N. Spencer, *Process-Oriented Guided Inquiry Learning*. Washington, DC, USA: Amer. Chem. Soc., 2008.

[30] C. Kussmaul, "Process oriented guided inquiry learning (POGIL) for computer science," in *Proc. ACM Tech. Symp. Comput. Sci. Educ.*, 2012, pp. 373–378.

[31] H. Hu and T. Shepherd, "Using POGIL to help students learn to program," *ACM Trans. Comput. Educ.*, vol. 13, no. 3, pp. 1–23, 2013.

[32] S. T. Ying, "Thinking in imperative or objects? A study on how novice programmer thinks when it comes to designing an application," in *Proc. IEEE Int. Conf. Eng. Technol. Educ. (TALE)*, 2019, pp. 1–7.

[33] "Engineering & engineering technology by the numbers," Amer. Soc. Eng. Educ., Washington DC, USA, Rep., 2019. Accessed: Dec. 16, 2021. [Online]. Available: https://ira.asee.org/wp-content/uploads/2021/02/Engineering-by-the-Numbers-FINAL-2021.pdf

[34] E. A. Stephan, D. R. Bowman, W. J. Park, B. L. Sill, and M. W. Ohland, *Thinking Like an Engineer*, 4th ed. London, U.K.: Pearson, 2017.

[35] J. Saldaña, *The Coding Manual for Qualitative Researchers*, 3rd ed. Los Angeles, CA, USA: SAGE, 2016.

[36] A. Strauss and J. M. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, CA, USA: SAGE Publ., Sep. 1998. Accessed: Dec. 16, 2021. [Online]. Available: http://www.amazon.co.uk/exec/obidos/ASIN/0803959400/citeulike-21

[37] N. Shrestha, T. Barik, and C. Parnin, "It's like python but: Towards supporting transfer of programming language knowledge," in *Proc. IEEE Symp. Visual Lang. Human-Centric Comput. (VL/HCC)*, 2018, pp. 177–185.

[38] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 691–701. Accessed: Dec. 16, 2021. [Online]. Available: https://doi.org/10.1145/3377811.3380352

[39] K. Steelman *et al.*, "Work in progress: The perception of computer programming within engineering education: An investigation of student attitudes, beliefs, and behaviors," in *Proc. ASEE Annu. Conf. Expo.*, 2020, pp. 1–8. Accessed: Dec. 16, 2021. [Online]. Available: https://www.asee.org/public/conferences/172/papers/29988/view

Briana Bettin received the B.S. degree in computer science from Michigan Technological University, Houghton, MI, USA, in 2014, the M.S. degree in human–computer interaction from Iowa State University, Ames, IA, USA, in 2016, and the Ph.D. degree in computer science from Michigan Technological University, in 2020.

She is currently employed as an Assistant Professor of Computer Science and jointly appointed as an Assistant Professor of Cognitive and Learning Sciences with Michigan Technological University. Her prior professional experience included front-end design and user experience services for a technology consulting firm. Her research interests focus on the intersection of experience design, education, and navigating an increasingly technological society.

Michelle Jarvie-Eggart (Member, IEEE) received the B.S. degree in environmental engineering, the M.S. degree in environmental policy, and the Ph.D. degree in environmental engineering from Michigan Technological University, Houghton, MI, USA, in 1996, 1999, and 2007, respectively.

She is a Registered Professional Engineer in the State of Michigan, USA, with a decade of experience working on environmental compliance and sustainability issues for the extractive industry. She is currently an Assistant Professor of Engineering Fundamentals and an Affiliated Faculty in Civil, Environmental, and Geospatial Engineering with Michigan Technological University. Her research interests include technology adoption, first-year engineering, and problem-based learning.

Kelly S. Steelman received the B.S. degree in aerospace engineering and the master's degree in mechanical and aerospace engineering from the Illinois Institute of Technology, Chicago, IL, USA, in 2002 and 2004, respectively, and the M.S. degree in human factors and the Ph.D. degree in psychology from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2006 and 2011, respectively.

In 2013, she joined the Department of Cognitive and Learning Sciences, Michigan Technological University, Houghton, MI, USA, as an Assistant Professor of Psychology and Human Factors, with affiliated appointments in Michigan Tech's Department of Mechanical Engineering–Engineering Mechanics and Department of Computer Science. She became an Associate Professor in 2019 and the Department Chair in 2020. Her research interests include technology acceptance and adoption, human performance in technology-mediated tasks, technology anxiety, computer training for older adults, and basic and applied human attention.

Dr. Steelman is a member of the Human Factors and Ergonomics Society.

Charles Wallace received the bachelor's degree in linguistics from the University of Pennsylvania, Philadelphia, PA, USA, in 1989, the master's degree in linguistics from the University of California at Santa Cruz, Santa Cruz, CA, USA, in 1992, and the Doctoral degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 1999.

He has been a member of the Computer Science Faculty, Michigan Technological University, Houghton, MI, USA, since 2000, and is also with the Department of Cognitive and Learning Sciences. He is currently an Associate Professor of Computer Science and an Associate Dean for Curriculum and Instruction with the College of Computing, Michigan Technological University. His research interests include computer science and software engineering education, accessibility of digital technology for underrepresented constituencies, ethics of computing, and applied formal methods.