On Students' Usage of Tracing for Understanding Code

Mohammed Hassan
University of Illinois
Urbana, IL, USA
mhassan3@illinois.edu

Craig Zilles
University of Illinois
Urbana, IL, USA
zilles@illinois.edu

ABSTRACT

Explain in Plain English (EiPE) questions evaluate whether students can understand and explain the high-level purpose of code. We conducted a qualitative think-aloud study of introductory programming students solving EiPE questions. In this paper, we focus on how students use tracing (mental execution) to understand code in order to explain it.

We found that, in some cases, tracing can be an effective strategy for novices to understand and explain code. Furthermore, we observed three problems that prevented tracing from being helpful, which are 1) not employing tracing when it could be helpful (some struggling students explained correctly after the interviewer suggested tracing the code), 2) tracing incorrectly due to misunderstandings of the programming language, and 3) tracing with a set of inputs that did not sufficiently expose the code's behavior (upon interviewer suggesting inputs, students explained correctly). These results suggest that we should teach students to use tracing as a method for understanding code and teach them how to select appropriate inputs to trace.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

KEYWORDS

tracing, explain in plain english, think-aloud

ACM Reference Format:
Mohammed Hassan and Craig Zilles. 2023. On Students' Usage of Tracing for Understanding Code. In *Proceedings of the 54th ACM Technical Symposium on Computing Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569741

1 INTRODUCTION

Students who are adept at providing high-level explanations of code tend to perform well at writing programs, implying that explaining code (which necessitates understanding) may be a precursor to writing code [12, 16]. On the contrary, explaining code at a low-level by restating the code line-by-line does not demonstrate an understanding of the purpose of the code [25]. "Explain in Plain English" (EiPE) questions (Figure 1), which require students to give a high-level explanation of code, are a common way to evaluate the skills of understanding and explaining code. A recent theory toward learning programming suggests that EiPE questions should be used towards learning common programming patterns [28] to help students read code written by others. In this paper, we investigate the process of how students understand code to explain its high-level purpose.

Write a short, high-level English language description of the code below. *Do not give a line-by-line description.*

Assume that the variable `x` is a `string`. *You can assume that the code compiles and runs without error.*

```
def f3(x):
    e = "aeiou"
    for c in e:
        if c not in x:
            return False
    return True
```

Figure 1: An example EiPE question with the high-level description of "Returns whether a given string contains all vowel letters."

While classifying the quality of student explanations has been extensively addressed in the literature [25], the thought process of students as they explain code has been seldom addressed. To the best of our knowledge, only Teague et al. has, as part of a broader study on characterising advancement levels of programming students, conducted think-alouds analyzing the thought-process of students explaining code [21–23]. Teague et al.'s main findings regarding EiPE questions (discussed in more detail in Section 2) were that advanced students can comprehend and make sense of code as they were reading it and, as a result, can explain code immediately after reading it. Less advanced students, on the other hand, needed to mentally execute (or trace) the code line-by-line multiple times first, to learn the relationship between inputs and outputs.

We also observed students using tracing to understand code, and this paper focuses on that practice. While the process of students solving tracing questions (e.g., find the output) has been well-studied and EiPE problems are well-regarded in the literature, how students use tracing to understand code (and, hence, solve EipE problems) has not been well studied. For example, Teague et al.'s studies included only two EiPE questions and those lacked important programming concepts, such as loops [21, 22]. In particular, our research questions are:

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569741

- How is tracing helpful for students to understand code and provide high-level explanations?
- What problems arise when students use tracing to understand code?

We find that tracing did help many of our research participants to correctly explain code that they could not explain without tracing, but three types of problems prevented tracing from aiding in some cases:

(1) Some participants were not aware that they could use tracing as a strategy to explain code. For these participants, the interviewer had to suggest to them to trace.
(2) Some participants did not trace the code correctly due to a misunderstanding related to the programming language, leading them to give an incorrect explanation based on their incorrect trace.
(3) Some participants did not choose a good set of inputs to sufficiently understand the general purpose of the given code. For these participants, the interviewer had to suggest useful inputs.

We begin, in Section 2, by describing prior work on the classification of the quality of student explanations of code and the different skills and advancement levels of programming. Then, in Section 3, we describe the research method of our study. Next, in Section 4, we describe the results of our study, starting with successful cases of students using tracing then the unsuccessful cases organized by the list above. Finally, we discuss the broader implications of this study, the limitations, conclusions, and future work.

2 RELATED WORK

Earlier work classified the quality of student-written explanations of code through SOLO's Taxonomy [2, 25]. Ideal explanations of code should be high-level and concise, which are classified as "relational" explanations. This is contrary to low-level, line-by-line explanations, which are classified as "multistructural" explanations. More recent work proposed an extended rubric for evaluating the quality of student explanations based on not only the level of abstraction but also factors such as correctness, ambiguity, and completeness [3, 24]. This helps differentiate from explanations that may be high-level but incorrect, confusing, or incomplete.

Teague et al. classified programming students' advancement levels based on the Neo-Piagetian framework as students solved EiPE questions, tracing questions [21–23], and fill-in-the-blank programming questions [23]. They found that more advanced novices can comprehend code as they read it by abstractly tracing with ranges/constraints rather than concrete values, demonstrating an understanding of the relationship between variables. When programmers are unable to use advanced strategies to understand code, they resort to tracing with concrete values [9, 22]. Teague et al. found that less advanced students rely on tracing to explain code, where they trace the code multiple times with multiple different inputs values, then infer the purpose of the code based on the input-output pairs obtained [21]. Detienne et al. found that even experts may resort to tracing upon reading unfamiliar code [9]. Programmers may also use tracing to help them find and fix bugs within programs as well [4, 17].

How students trace code has been well-studied in the context of tracing (e.g., find the output) questions, which do not require students to select inputs. For example, there are studies analyzing students' drawings/notes as they trace [6, 14] and tools made to scaffold tracing [5, 19]. On the other hand, how tracing can be used to understand and explain code at a high-level is not well-investigated.

Unlike traditional tracing problems, tracing for understanding code requires students to select inputs. To our best knowledge, how students choose inputs to understand others' code is not well investigated. The most similar work is that of students designing test cases for their code, where students know the goal of their program. Students can have an oversimplified view of the input-output relation of programs, often writing test cases that only cover expected, typical-case scenarios and do not account for hidden, unexpected situations that are likely to reveal bugs [10, 13]. Input-output exercises are also known to be beneficial for code-writing questions. Since students often misunderstand code-writing question prompts and work toward solving the wrong problem [15], Prather et al. [20] and Denny et al. [8] asked students to find the output for corresponding inputs pertaining to the question prompt before writing code. These input-output exercises helped students develop meta-cognitive awareness to correct their understanding of the code-writing prompt.

3 METHODS

We conducted a series of think-aloud interviews to observe how students (the participants) solved EiPE questions and how they used tracing toward solving the problems. We followed the protocol of Ericsson et al. for conducting think-aloud interviews, where we asked participants to verbalize only their thought process without translating it for our benefit to minimize fatigue and third factors [11].

The 15 participants were traditional-aged undergraduate students (9 males, 6 females, age range 18-23) who had completed an introductory programming course in Python for non-technical majors during the Fall 2021 semester. The course instructor has taught how to solve EiPE questions (e.g., showing examples of ambiguous, low-level answers to avoid). The EiPE questions administered during the course differ from the questions used for this study. With IRB permission, we recruited these participants through an email sent to the class roster. Each interview was approximately one hour, and participants were compensated with a $15 gift card.

For this paper, we focused our analysis on 11 of these interviews. The other interviews did not exhibit behaviors relevant to this work (i.e., no usage of tracing). We present excerpts from 7 of the 11 interviews due to space limitations. This study was conducted in two phases. For the first phase, we asked participants to think-aloud as they solved EiPE questions. If they were silent for more than 2-3 minutes, we reminded them to think-aloud. The second phase was conducted in a similar fashion, but if participants were stuck on solving problems for more than 2-3 minutes (appearing to make no progress), we additionally asked them to trace the code to determine if this helped them to explain the code. If participants were still unable to explain the code even after tracing, we suggested specific inputs to use when tracing the code. We conducted the second phase

after we noticed the prevalent behavior of participants resorting to tracing. All participants were given approximately equivalent sets of 12 Python-language EiPE questions; question order was varied. For each question, the participants were provided with the following instructions:

> Write a short, high-level English language description of the code in the highlighted region. Do not give a line-by-line description.
> Assume that the variable x is a <datatype, e.g., string, int, etc>. You can assume that the code compiles and runs without error.

The interviews were recorded over Zoom due to COVID-19, then transcribed and analyzed independently by two researchers, who met to discuss differences in interpretations. We inductively coded the data, documenting: 1) the inputs participants chose for tracing the function (e.g., did they choose inputs that helped them understand the function?), 2) whether they traced correctly (e.g., did they follow correct programming language rules like correct order of execution?), 3) the quality of the participant's explanation of the function (e.g., is it high level and relational, per SOLO's Taxonomy [2]?). The two researchers independently identified themes from the codes and reconciled differences to produce a final list of themes.

4 RESULTS

We found that tracing can help some participants solve EiPE questions. Furthermore, we identified three ways participants can fail to use tracing to help themselves solve EiPE questions.

(1) They do not consider tracing the code when it could help them. We had participants that were struggling to correctly explain the function, but when prompted by the interviewer to trace the code, they then correctly explained the code.
(2) They failed to correctly trace the code, not following appropriate programming language rules, such as the order-of-execution, syntax rules, built-in function definition rules, and so on.
(3) They failed to choose a set of inputs that exposed enough of the function's behavior to understand its purpose (e.g., different paths of if-statements, vary number of loop iterations, empty vs non-empty data structures, non-null).

4.1 Tracing can help Participants Solve EiPE Questions

The concrete nature of tracing code can help learners understand the execution behavior of the code. For example, Participant 1 gave an incorrect explanation of the function in Figure 1. The interviewer notified the participant that their explanation was incorrect. Then, the participant chose to trace the code to check their explanation. As they partially traced the loop, they recognized its overarching pattern and corrected their explanation of the code.

> **Participant 1:** *Writes incorrect explanation "This function returns False if string x does not have any vowels. If string x does have vowels it returns True" for the question in Figure 1*
> *(Interviewer notifies participant that their answer is incorrect)*

```
def f1():
    x = 0
    c = -1
    z = -1
    while z != 0:
        z = int(input("Give me a number: "))
        x += z
        c += 1
    return x / c
```

Figure 2: A correct, high-level description of this function is "Returns the average of all given input numbers, stopping at 0."

> **Participant 1:** *Tries input string "bceray"*
> **Participant 1:** for each of these values in e, for "aeiou" in e, if c not in x, return False. If (pause)
> **Participant 1:** Ok so for c in e, ... So it starts at 'a', so for 'a' in e, ... if 'a' is not in x, it returns False. Then it goes back up.
> **Participant 1:** So it checks to see if each the (pause) if a e i o but it goes in order, ... (pause)
> **Participant 1:** When does it return true? Ok it returns true if we (pause) ...
> **Participant 1:** If a e i o or u is not in x, it returns False? ...
> **Participant 1:** Oh, if all 5 vowels are in x it returns true, else it returns false *(correct explanation)*.

4.2 Some Participants Who Struggled to Explain Functions Did Not Consider Tracing

Some participants that struggled to explain functions did ***not*** consider tracing. Instead, they appeared stuck (e.g., were silent for more than two minutes) or repeatedly explained the function incorrectly without attempting to trace. In those cases, the interviewer suggested tracing, which led some participants to correctly explain the function. For example, participant 11 appeared stuck on explaining the function in Figure 2. The interviewer suggested to trace the code. The participant then chose inputs and traced the code, but appeared confused about variable c being initialized to -1. As they continued to trace more, they then independently resolved their own confusion and correctly explained the function.

> **Participant 11:** *(Appears confused, not giving any explanation)*
> **Interviewer:** Try out some numbers and see what happens. Maybe it might help you.
> **Participant 11:** Sure. I can try 3. ... Because z is not equal to 0, you're gonna do x plus equals 3 which means x is equal to 3. And then wait. Yea. My confusion here is why is z set to -1?
> *(Interviewer suggests to the participant to continue tracing)*
> **Participant 11:** So x would be equal to 3, and then c would have 1 added to it, making it 0. It would return

Assume that the variable x is a list of ints.

```
def Rf17(x):
    for w in x:
        if w % 2 == 0:
            return w
    return -1
```

Figure 3: A correct, high-level description of this function is "Returns the first even number from a given list."

3 divided by 0, gives an error.
Participant 11: Maybe if I try another number. If its 5 then x is equal to 8. z is equal to 1. Return ...
Participant 11: If we try another number, let's say 2 ... So c is the count minus 1. And then x is the sum of all the numbers you input. So the sum (pause) oh, it is the average.

4.3 Some Participants Traced Incorrectly

Some participants did not trace the code correctly because they did not follow the appropriate programming language rules (e.g., incorrect order-of-execution). Participants who traced incorrectly gave incorrect explanations of the function. In these two examples, Participant 3 gave incorrect explanations of the functions. The interviewer then prompted them to trace the function with an input in an attempt to assist the participant. The participant responded with incorrect outputs and did not resolve their incorrect explanations. This participant appeared to mistakenly believe that `return` statements do not terminate a function's execution (e.g., treating `return` like `print` [1]), which interfered with their ability to trace the code.

Participant 3: *Writes incorrect explanation "Returns False if there are no vowels in string x, or returns True" for question in Figure 1*
Interviewer: Do you want to try a 3 letter string and walk through it?
Participant 3: *Tries input string "aep"*
Participant 3: It would return True, True, and False.

Participant 3: *Writes incorrect explanation "returns the even integers of the list x" for question in Figure 3*
Interviewer: If the input to the function was the list `[2,3,4]`, what would the output be?
Participant 3: 2 and 4, *(incorrect, output is 2)* because they are divided by 2 and remainder is 0.

4.4 Some Participants Did Not Choose a Good Set of Inputs

Some participants traced the function with a limited set of inputs that did not expose the general behavior of the function and, as a result, could not correctly explain the function. These types of incorrect explanations often gave a correct lower-level fact about

Assume that the variable x is an int.

```
def f19(x):
    o = 0
    while x > 0:
        if (x % 10) % 2 == 0:
            o += 1
        x //= 10
    return o
```

Figure 4: A correct, high-level description of this function is "Returns the number of even digits in a given number."

the function but does not represent the high-level purpose of the function.

For example, Participant 6 was unsure how to explain the function in Figure 4 initially, so they chose to trace to try to understand the behavior of the function. They correctly traced with many different inputs (e.g., 1000, 1500, 1700, etc), but none of the inputs had even digits besides 0. As a result, they mistakenly believed that the function counts the number of zeroes in a given integer.

Participant 6: *(Traces input 133 for question in Figure 4)*
Participant 6: I'm thinking it checks the numbers of tens, hundreds, thousands.
Participant 6: *(Traces input 1000, 1500)*
Participant 6: I'm not sure what this `if` statement checks but I know that after it does this checking, it divides this number by 10, gives you the remainder. This `if` statement is the key *(pause)*.
Participant 6: *(Traces input 15, 0, 1700)* I'm checking different numbers for x. ... I was thinking it gives a count of the number of zeroes. ...
Participant 6: Ok the theory I have is that it counts the number of zeroes. ...
Participant 6: *(Writes explanation "returns the number of zeroes that appear in a given number," incorrect as it is functionally incomplete)*

As another unsuccessful example, participant 10 gave an initial, incorrect explanation. After the interviewer notified the participant that their explanation was incorrect, they chose to trace the code with only the empty list as the input which did not demonstrate the execution of the 'for each item in list' loop. Although they traced correctly, they were unable to correct their explanation.

Participant 10: *(Wrote incorrect explanation "returns even integers from a list" for question in Figure 3)*
(Interviewer notifies participant that explanation is incorrect)
Participant 10: *(chose to trace input `[]`)* If it's an empty list the output is -1 *(Correct trace)*.
(Participant revises explanation to "returns even integers and if it's empty it returns -1," still incorrect)

For complex functions that have at least one conditional and loop, successful participants selected specific sets of inputs to expose the general behavior of the function. These participants started

Assume that the variable `x` is a `string`.

```
def f7(x):
    o = ""
    for i in range(len(x)-1, -1, -1):
        o += x[i]
    return o
```

Figure 5: A correct, high-level description of this function is "Reverses a given string."

tracing initially with arbitrary values of the appropriate data type. Afterwards, these participants narrowed down to values holding specified characteristics (e.g., fulfilling an `if` condition a specified number of times within a `for` loop) to better understand the function.

For example, participant 12 immediately chose to trace with inputs for the function in Figure 4. They appeared to choose round numbers of 2-3 digits that had even and odd digits (e.g., 400, 120, 20, etc). They seemed confused about the purpose of the function but continued to repeat the same set of inputs. Finally, they tried a larger number, 2398, with 4 digits that had 2 even digits, 2 odd digits, and no 0 digits. Then, they gave a correct explanation of the function only after tracing with this larger input.

For cases where participants were unable to choose a good set of inputs, the interviewer would provide the participant with a set of inputs as assistance. This always led participants to successfully explain functions, assuming the participant traced correctly.

For example, participant 4 began by reading the code in Figure 1 and provided an incorrect explanation of the function. After the interviewer notified them that their explanation was incorrect, they traced the code with two of their own self-selected inputs. After each of their own traces, they revised their explanation to be a correct lower-level fact about the function, but was still not representative of the complete, high-level purpose. For instance, "the function will return True if the string input was 'aeiou,"' but that is not the high-level purpose of the function. Afterwards, the interviewer provided the participant with two inputs to help them to better understand the function. The first provided input led the participant to revise their explanation to be closer to the correct answer, and the second input led the participant to revise their explanation to be the correct high-level purpose.

> **Participant 4:** *(Wrote incorrect explanation "Returns* `True` *if 'a' is present in string x" for question in Figure 1)*
> *(Interviewer notifies participant that answer is incorrect)*
> *(Participant chooses to trace with input string 'lollipop')*
> **Participant 4:** If 'a' is not in lollipop, it would return `False`, but if 'a' is, then it'd go again, then it'd go again (pointing mouse cursor throughout `for` loop).
> **Participant 4:** Oh wait it wouldn't go again, oh my gosh (pause).
> *(Writes incorrect explanation that is closer to the correct answer, containing a correct lower-level fact about the function "Returns* `True` *if string x equals 'aeiou"')*
> **Participant 4:** For c in e, if c is not in x, its (a) not, it would return `False`, but if this was, if this was
> *(Participant chooses to trace with input string 'aeiou')*
> **Participant 4:** ... if c is not in x, it (a) is, for c in e, if c not in x, it (e) is ..., and then after it does 'u', it would go out and return `True`.
> *(Interviewer notifies participant that explanation is incorrect. Interviewer suggests to trace with the input string 'helloaeiou')*
> **Participant 4:** for c in e, if c is not in x ..., so then it'd have to be present. ...
> *(Participant revises explanation to "Return False if 'aeiou' is not present in string x, else True", which is incorrect since order of characters does not matter, but even closer to the correct answer)*
> *(Interviewer notifies explanation is incorrect, and suggests to trace with input string 'hell**ao**eiou')*
> **Participant 4:** It goes through and it (points mouse at 'aeiou') is still is present. ... *(Participant revises to correct explanation "Returns False if the characters in 'aeiou' is not present in x, else True")*

For simple functions (e.g., functions that lack if statements), our participants always understood the function by tracing with only one input. This may suggest that simple functions are easy to explain with tracing because participants can easily guess good inputs to understand those functions. Almost any input value of the correct data type would be sufficient for participants to understand the function. For example, for the function in Figure 5, any one string that is not a palindrome would be sufficient to test the behavior of the `for` loop (e.g., most names/words meet this criteria). All of our participants chose a string that met this criteria on their first attempt and consequently understood the function through tracing, even including participants who struggled to choose good inputs on more complex problems.

5 DISCUSSION

To answer RQ1, *How is tracing helpful for students to understand code and provide high-level explanations?*, our successful participants made use of tracing in one of two ways:

(1) Participants performed a partial trace of the code and, through this process, they recognized the overarching pattern. They then provided a high-level explanation (e.g., participant 1 in Section 4.1).
(2) Participants performed several complete traces of the code. After some complete traces, these participants iteratively revised their explanation based on the observed input-output relationships. This process repeats until the participant has chosen a sufficient set of inputs to expose the high-level purpose of the function (e.g., participant 12 in Section 4.4).

One potential reason some participants only needed a partial trace to understand code in some problems may be due to the differing difficulties of problems. Some participants used partial traces only on problems that utilized loops, appearing to recognize the pattern of the remaining iterations of the loop mid-trace. Cunningham et al. retrospectively interviewed students about their notes

on tracing questions and also found that some students performed a partial trace [7]. Our partial trace finding differs from Teague et al.'s studies [21–23] in that Teague et al.'s successful participants who traced always performed complete traces rather than a partial trace, not appearing to make sense of the overarching pattern of the code in the middle of their trace. Rather, they inferred the purpose of code using multiple input-output pairs obtained after completing multiple traces. Teague et al.'s EiPE questions lacked loops, potentially implying that this partial trace strategy may be applicable only to loops.

Like Teague et al. [21–23], we also found that some participants needed multiple complete traces to explain code, but with our more complex problems, we add further insight to Teague et al.'s finding, in that many of our participants performed an iterative process where they revised their answer in between multiple traces. Some of our participants began with providing a correct fact that does not capture the complete, high-level purpose, and as they traced with more inputs, they revised their answer to be closer to the complete, high-level purpose. Successful participants correctly explained the code after multiple iterations.

To answer RQ2, *What problems arise when students use tracing to understand code?*, we found three:

(1) Some participants were not aware that they can use tracing as a strategy to explain code.
(2) Some participants did not trace the code correctly due to a misunderstanding related to the programming language.
(3) Some participants did not choose a good set of inputs to sufficiently understand the general purpose of the given code.

Since many participants needed the suggestion to trace to successfully solve EiPE questions, our results suggest that we should teach introductory programming students to use tracing as one of the strategies to understand code. Encouraging the use of tracing for understanding is reasonable developmentally because the ability to trace often precedes the ability to read (understand and explain) code [14]. Contrary to some of our participants being unaware of using tracing, Teague et al. [21–23] found that all participants in their study who initially struggled to explain independently chose to trace, as tracing questions always preceded EiPE questions in their studies, likely priming students to trace first.

Some participants traced incorrectly, demonstrating misunderstandings related to the programming language and, as a result, gave an incorrect explanation consistent with their incorrect trace. These incorrect explanations are still high-level (e.g., "returns the even integers of the list x" in Section 4.3), similar to the finding of Murphy et al. [18] of students who abstract code incorrectly. We hypothesize that these participants abstract incorrectly due to misunderstandings of program execution behavior. Future work is needed to confirm whether it is a misconception solely about the programming language or additionally on the process of explaining code (e.g., can there be students who trace correctly with sufficient inputs but give incorrect, high-level explanations?).

We found that, for participants who needed to trace multiple times, some did not choose a sufficiently useful set of inputs to understand the code. These participants gave incomplete, high-level explanations (e.g., participant 6 in Section 4.4). After we provided inputs to guide these participants, they explained the code correctly. Since they needed this guidance, we should teach students how to choose helpful inputs to understand code. Wrenn et al. [26, 27] found that students who write test cases for programming problems assume non-existent functionality and non-existent constraints on possible inputs. We observe a similar example with participant 6 where they seem to conclude that the function counts decimal places immediately after the first input, unlike successful participants who expressed confusion after their first (often few) inputs rather than making early conclusions. Participant 4 also quickly assumed the code's functionality immediately after their first input. Participant 10 chose only the empty list as an (overly constraining) input. As student test cases also tend to miss rare implementation bugs [13, 27], a potential recommendation is to teach students to avoid making early assumptions about code until they have tried a sufficiently diverse set of inputs.

6 LIMITATIONS

Our data may not be representative of students in general. As typical of most qualitative studies, the sample size is relatively small due to the large amounts of data to analyze per interview and time commitments to coordinate and perform interviews. Also, our participants are self-selected, which might mean that they have higher than average self-confidence as they chose to participate in the study and may be among the higher performers of the course.

7 CONCLUSIONS & FUTURE WORK

In this paper, we have presented a study on how introductory programming students use tracing to understand and explain the high-level purpose of functions. We found that while tracing can be helpful toward understanding functions, there were three types of issues that prevented students from using tracing to understand code. The first issue was students not considering using tracing when it could help them understand the function. For these students, the interviewer suggested to them to trace, and they were then able to understand the code (assuming they traced correctly). The second issue was students not tracing the code correctly, demonstrating misunderstandings related to the programming language. The third issue was students not selecting a good set of inputs that would expose the general behavior of the function (e.g., not selecting even digits for a function that counts the number of even digits).

These findings informs us of how we can help students understand and explain functions, by 1) suggesting to students to trace if they struggle to understand the function and 2) teaching students how to select useful input arguments to trace to understand the function. More research is required on how to teach students to select appropriate inputs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2121424. This work was supported in part by Mohammed Hassan's SURGE and graduate college fellowships at the University of Illinois.

REFERENCES

[1] Austin Cory Bart, Teomara Rutherford, and James Skripchuk. 2020. Evaluating an Instrumented Python CS1 Course.. In *CSEDM@ EDM*.

[2] John B. Biggs and K. F. Collis. 1982. *Evaluating the quality of learning : the SOLO taxonomy (structure of the observed learning outcome) / John B. Biggs, Kevin F. Collis.* Academic Press New York. xiii, 245 p. : pages.

[3] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[4] Woei-Kae Chen, Chien-Hung Liu, and Hong-Ming Huang. 2017. Software Debugging Patterns for Novice Programmers. (2017).

[5] Chih-Yueh Chou and Peng-Fei Sun. 2013. An educational tool for visualizing students' program tracing processes. *Computer Applications in Engineering Education* 21, 3 (2013), 432–438.

[6] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 164–172.

[7] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice rationales for sketching and tracing, and how they try to avoid it. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 37–43.

[8] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A closer look at metacognitive scaffolding: Solving test cases before programming. In *Proceedings of the 19th Koli Calling international conference on computing education research*. 1–10.

[9] Françoise Détienne and Elliot Soloway. 1990. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies* 33, 3 (1990), 323–342.

[10] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 171–176.

[11] K. Anders Ericsson and Herbert A. Simon. 1980. Verbal reports as data. *Psychological Review* 87 (1980), 215 – 251.

[12] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* (2022), 1–29.

[13] David Ginat. 2007. Hasty design, futile patching and the elaboration of rigor. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*. 161–165.

[14] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[15] Dastyni Loksa and Amy J Ko. 2016. The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM conference on international computing education research*. 83–91.

[16] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ACM, 101–112.

[17] Rifat Sabbir Mansur, Ayaan M Kazerouni, Stephen H Edwards, and Clifford A Shaffer. 2020. Exploring the Bug Investigation Techniques of Intermediate Student Programmers. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.

[18] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English'questions: implications for teaching. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 385–390.

[19] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM conference on international computing education research*. 2–11.

[20] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM technical symposium on computer science education*. 531–537.

[21] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]*. Australian Computer Society, 87–95.

[22] Donna Teague and Raymond Lister. 2014. Blinded by their Plight: Tracing and the Preoperational Programmer.. In *PPIG*. 8.

[23] Donna Teague and Raymond Lister. 2014. Longitudinal think aloud study of a novice programmer. In *Conferences in Research and Practice in Information Technology Series*.

[24] Renske Weeda, Cruz Izu, Maria Kallia, and Erik Barendsen. 2020. Towards an Assessment Rubric for EiPE Tasks in Secondary Education: Identifying Quality Indicators and Descriptors. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.

[25] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P K Ajith Kumar, and Christine Prasad. 2006. An Australasian study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Eighth Australasian Computing Education Conference (ACE2006)* (2006).

[26] Jack Wrenn and Shriram Krishnamurthi. 2021. Reading Between the Lines: Student Help-Seeking for (Un) Specified Behaviors. In *21st Koli Calling International Conference on Computing Education Research*. 1–6.

[27] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (2021).

[28] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.