Some Thoughts on Designing Eye Movement Studies for Novice Programmers

Raymond Lister
School of Computer Science
University of Technology, Sydney
Sydney NSW Australia
Raymond.Lister@gmail.com

ABSTRACT

I first describe my three-stage model of how novices understand code. In the first stage, the novice cannot trace code. In the second stage, the novice has mastered tracing, but, crucially, that is the only skill they have mastered. It is only when novices reach the third stage that they begin to reason about code in a more general, abstract way. Most programming instructors mistakenly assume that all students begin at the third stage. Having described the three-stage model, I then explore implications of the model for the design of eye movement studies. I also provide some pieces of code that would make for interesting eye movement studies.

CCS CONCEPTS

Social and professional topics → Computing education

KEYWORDS

Novice programmers. Eye movement. Eye scanning.

ACM Reference format:

Raymond Lister. 2022. Some Thoughts on Designing Eye Movement Studies for Novice Programmers. In *Proceedings of the Tenth Workshop on Eye Movement in Programming (EMIP 2022). ACM, New York, NY, USA, 8 pages.*

1 INTRODUCTION

My twenty years of studying novice programmers has been driven by one simple research question: *What code-related skills precede code writing*? Here, I will not give a detailed account of how those twenty years unfolded. Such an account can be found elsewhere [4, 5] and that account will in turn lead any interested readers to the papers I wrote over those years.

In this paper, I will begin by summarizing the results from my twenty years of research on novice programmers. I will then make some suggestions on the design of eye movement studies of novice programmers, based upon that research. Finally, I present some pieces of code I have used in my research which I believe might produce interesting results in an eye movement study.

2 TRACING CODE AND EXPLAINING CODE

2.1 Tracing Code

The Leeds Working Group collected data from over 600 introductory programming students, spread across 12 institutions in 7 countries [1]. The working group found that most students in the participating institutions could not trace code reliably. That is, given some code and either input data or initial values for the variables, most students at the end of their first semester of learning to program could not reliably manually execute (or "desk check") the code using pen and paper.

2.2 Explaining Code

Shortly after the Leeds Group, participants in the BRACElet project set out to answer a question that followed obviously from the Leeds Working Group study – *apart from tracing, are there other precursor skills to code writing?* To address that question, BRACElet introduced a new type of question, to explore if students could read and understand code. Figure 1 provides an example of such a question. This new type of question was called the "Explain in Plain English" question. In retrospect, this was a poor choice of name; "Explain in Plain Language" would have been better. In this paper, I will use an even simpler name, "explanation question".

In plain English, explain what the following segment of Java codes does:

```
bool bValid = true;

for (int i = 0; i < iMAX-1; i++)
{
    if (iNumbers[i] > iNumbers[i+1])
       bValid = false;
}
```

Figure 1. The first explanation question studied in the BRACElet project [2, 14].

While it is not obvious in Figure 1, students were not being asked to provide a line-by-line description of the code. Instead, students were expected to provide a summary of the overall computation performed by the code. For the code in Figure 1, a suitable explanation would be something like "it checks to see if the array is sorted".

The first two papers published by BRACElet [2, 14] described the results and conclusions from this first round of work by BRACElet. One of the results was that students who gave a suitable answer to the explanation question in Figure 1 tended to perform better on a code writing task. In the conclusion of one of those first two BRACElet papers [2], we speculated:

> *In our view, students who cannot read a short piece of code and describe it … are not intellectually well equipped to write similar code.*

2.3 Tracing + Explaining → Writing

BRACElet participants then went on to empirically study the relationship between tracing, explaining, and writing code. They found that code tracing questions alone did not correlate well with student scores on code writing, nor did explanation questions alone correlate well with student scores on code writing. However, the combination of student scores on code tracing and code explaining did correlate well with code writing [6].

The graph in Figure 2 is from a subsequent study by the BRACElet project, which confirmed the relationship between code tracing, code explaining and code writing [13]. As shown in that figure, a combination of student scores on tracing and explaining tasks accounted for 66% of the variance in student scores on code writing tasks. The two ovals in Figure 2 highlight that no student who performed poorly on the combination of tracing and explaining performed well on code writing, and no student who performed well on the combination of tracing and explaining performed poorly on code writing.

3. A THREE STAGE MODEL

Further work led me to propose a three-stage model of the development of novice programmers [3]. In my early papers on the three-stage model, I used names for each stage that were based on neo-Piagetian theory – sensorimotor, preoperational and concrete operational. However, I found those neo-Piagetian stage names troubled many people, so I have since adopted new stage names, used below, but the characteristics of each stage are unchanged from my early papers that used the neo-Piagetian names.

3.1 Stage 1: Pre-Tracing

In the initial pre-tracing stage, novices cannot reliably trace code. There are several reasons why novices can struggle to even trace code. Perhaps the best-known reason is that novices at this stage have misconceptions of how programs work. For example, a novice might think that the assignment statement "`x = y`" entangles those two variables so that any subsequent update to one variable also updates the other variable. See appendix A of Juha Sorva's thesis for a catalogue of over one hundred misconceptions [9].

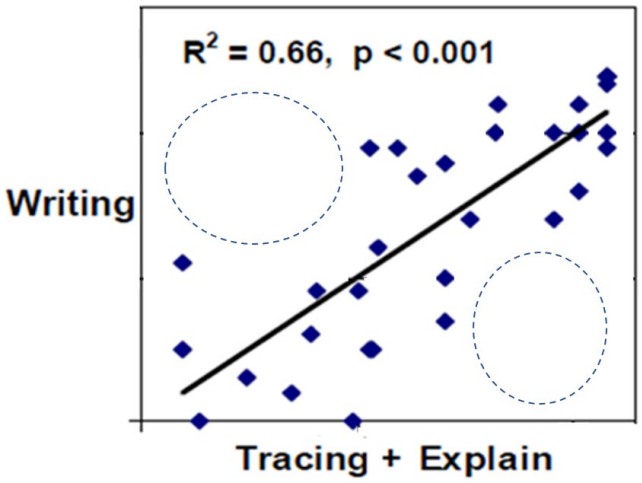

Figure 2. A graph from Venables, Tan and Lister (2009). Some details of the original graph, such as the axis scales, have been omitted here, for simplicity.

When novices at the pre-tracing stage are required to write code, they exhibit a haphazard approach, or they resort to Cargo Cult Programming [15], or Voodoo Programming [18], where they copy at least part of the solution from other sources, such as books or websites, without truly understanding the code they copy. If the problem given to such a novice is small, they may stumble their way to a solution, or at least to a buggy partial solution, but have no real understanding how the code works. Many experienced teachers will have had the experience of asking a student why they have placed a particular line of code in their program, which is often a superfluous or bizarre line of code, and the student cannot offer a reasonable explanation for why they have that line. Voodoo Programming is an especially apt term, because for the pre-tracing novice a line of code can be a mysterious, magic spell.

When a teacher sees a student trying to program this way, the most common response is to tell the student to not copy code, and to write code in a more principled way. However, to do so is to treat the symptom, not the underlying problem. The student in the pre-tracing stage is yet to learn to reason about code in a principled way.

3.2 Stage 2: Tracing (inductive)

By the second stage, the tracing stage, the novice has a sufficiently coherent and systematic understanding of code that the novice is capable of reliably tracing code. However, and crucially, the tracing stage novice often cannot abstract beyond the code itself. The only way that a tracing stage programmer can reason about a piece of code is by induction; that is, by tracing the code. When attempting to explain what a piece of code does, the tracing stage programmer (1) generates a set of initial variable values, (2) traces the code, and then (3) attempts to infer the function of the code by comparing the initial and final values.

Tracing stage novices tend to use that same inductive approach when attempting to write and debug their own code, in a process sometimes called "programming by permutation" [16], or "shotgun

debugging" [17]. That is, the tracing stage novice will often trace their buggy code with specific values, and then make what is often a myopic patch. That patch may "fix" the code for the specific initial values just used in the trace, but the patch may not address the general bug. Tracing stage novices may make a series of such myopic patches, without abstracting to a general understanding of the fundamental problem with their code

Perhaps the primary contribution of the three-stage model is the explicit identification of the tracing stage. For teachers who are not aware of the literature on code explanation, it can be difficult to accept that some students who can trace code cannot also reason about code in a more abstract way. Consider, for example, Thomas, Ratcliffe, and Thomasson [11], who wrote the following after trying to help their novices to make effective use of diagrams:

> *Providing ... what we considered to be helpful diagrams did not significantly appear to improve their understanding ... This was completely unexpected. We thought that we were 'practically doing the question for them'.* [p. 253]

As with the previous stage, when a teacher sees a tracing stage student trying to program this way, the most common response is to tell the student to write code in a more principled way. Once again, however, to do so is to treat the symptom, not the underlying problem. The tracing stage student is not capable of writing code in a more principled way.

In this paper, I will not describe the neo-Piagetian theory underlying the three-stage model. Consequently, the reader may doubt the existence of the tracing stage, or at least be skeptical that a novice can remain in the tracing stage for a protracted period. For more about the neo-Piagetian aspects of the three-stage model, see earlier papers by me [3, 4, 5] and see the collection of papers in Donna Teague's thesis-by-publication [10] for case studies of novices in the tracing stage.

3.3 Stage 3: Post-Tracing (deductive)

It is only at the third stage, the post-tracing stage, that a student begins to reason about programs the same way as their teacher. That is, post-tracing stage novices, like their teachers, begin to reason about code deductively, by simply reading the code, and/or by relating code to diagrammatic representations of operations on data structures. This is the stage when a novice can explain code just by reading it. It is also the stage where novices begin to show a coherent, purposeful approach to writing code.

The principal factor underlying the decades-long tradition of poor outcomes in the teaching of programming has been the false assumption by teachers that novices begin at the third stage, or that novices skip quickly from the pre-tracing stage to the post-tracing stage. This false assumption leads teachers to talk about code in abstract terms before their students are ready to understand code in abstract terms.

3.4 Overlapping Waves

From the above description of the three stages, the reader might incorrectly infer that novices progress through the stages in a quantum-like way, working at one stage before suddenly making a leap to the next stage. In this paper. I initially described the development of the novice programmer as three distinct phases to keep the introduction of these ideas simple, but the reality is more complex, as novices exhibit an evolving mix of the three stages. This concept of "overlapping waves" of stage progression is illustrated in Figure 3.

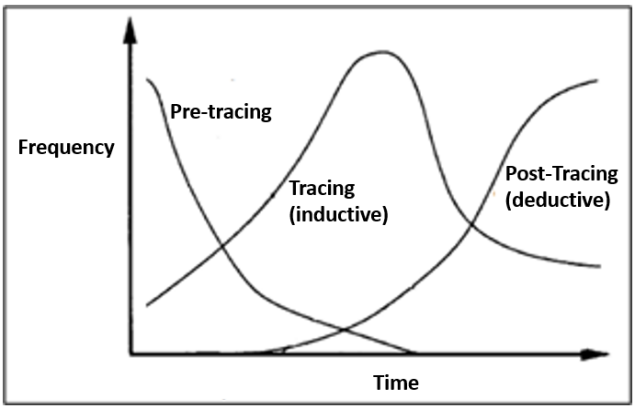

Figure 3. The Concept of Overlapping Waves.

In the early period of learning to program, the novice transitions progressively from pre-tracing to tracing as the novice steadily eliminates misconceptions and becomes more systematic about conducting a trace with pen and paper.

The progressive transition from tracing to post-tracing occurs as the novice steadily acquires programming plans, often called "schemas". See chapter 4 of Sorva's thesis for a review of schemas in a programming context [9]. A novice programmer who is primarily at the tracing stage has not acquired many programming schemas, so such a novice relies heavily on their tracing skill to reason about code. As the novice acquires more schemas, the novice relies more on those schemas to reason about code, and less on tracing. When the novice becomes primarily reliant on reasoning via schemas, and only occasionally resorts to tracing, the novice has reached the post-tracing stage.

4. SOME THOUGHTS ON EYE MOVEMENTS

I now present some thoughts on the implications of my three-stage model for eye movement studies of novices answering code explanation questions. Central to my thinking is the belief that the most interesting novice to study with eye movement data is the novice in transition from the tracing stage to the post-tracing stage.

4.1 Pre-test for Code Tracing Ability

I doubt that studying eye movement data will reveal anything new about the misconceptions of the pre-tracing novice. I therefore suggest a short test of tracing skill to screen-out pre-tracing novices from an eye movement study.

An exception to this suggestion might be the case where your interest is in developing a tutoring system that can use eye movements to assess the current developmental stage of a novice, but I suspect having novices trace simple pieces of code is a more direct and effective means of detecting the pre-tracing novice.

4.2 Explicitly Test Code Writing

The goal is to have novices learn to write code. Consequently, studies of explanation questions should try to connect eye movement data to code writing ability. It has been common practice to characterize novices by how much programming experience they have (e.g., number of weeks of learning to program). These are noisy proxies for estimating code writing ability. I think we need to test novices more directly and precisely on their code writing ability, by giving them one or more specific code writing tasks.

4.3 Study Multiple Explanation Questions

I think a comprehensive eye movement study with explanation problems needs to present each novice with several explanation problems of varying difficulty.

4.4 Ignore Explanations Derived by Induction

The novices who are in transition from the tracing stage to the post-tracing stage will attempt to answer some of the explanation questions by deduction (i.e., simply reading the code) and some questions by induction (i.e., by tracing). I am skeptical that eye movement data is useful when a novice answers a question by induction, since such a novice will (1) systematically look at lines of code in the order the lines are executed, and (2) will avert their eyes from the code for much of the time, to record changing variable values on paper.

One option is to simply forbid the participating novices from resorting to tracing. I think that option is problematic as it will lead to guessing and there is probably little value in the eye movement data of a novice who has guessed.

Another option is to allow novices to trace if they wish (without communicating any discouragement to them), but then discard their eye movement data for that question and focus on the eye movement data for each novice on each explanation question where the novice has answered by deduction.

I know that participants can be hard to recruit, and eye movement data is expensive to acquire, so researchers conducting an eye movement study will be reluctant to accept my recommendation and discard any eye movement data. However, if the novice of interest to you is the same type of novice of interest to me – the novice in transition from the tracing stage to the post-tracing stage – then eye movement data from other types of novices adulterates the data from the interesting novices.

While I find most interesting the novice in transition from the tracing stage to the post-tracing stage, it is legitimate to study less advanced novices. Perhaps my recommendation can be framed neutrally, as follows – either (1) be explicit about the type of novice you want to study, and screen out other types of novices, or (2) collect data from all types of novices but analyze separately the data for each type of novice.

4.4 What to Compare?

Figure 4 shows some options for comparing the eye movement data from novices on a specific explanation question. In that figure, it is not clear what it means for a novice to have answered a "low" or "high" number of explanation questions correctly by deduction. I leave it to anyone who conducts an eye movement study to disambiguate "low" and "high" for themselves. In Figure 4, the comparison of most interest to me are the novices who fall into the top right quadrant or the bottom right quadrant. That is, of most interest to me are the novices who answered most of the full set of explanation questions correctly, by deduction, but who split into those who answered a specific question correctly (bottom right quadrant) and those who did not answer correctly (top right quadrant). Also of interest to me are the novices who fall into the bottom left quadrant or the bottom right quadrant; that is, novices who answered a specific question correctly, but who split into those who scored high on the full set of explanation questions and those who did not.

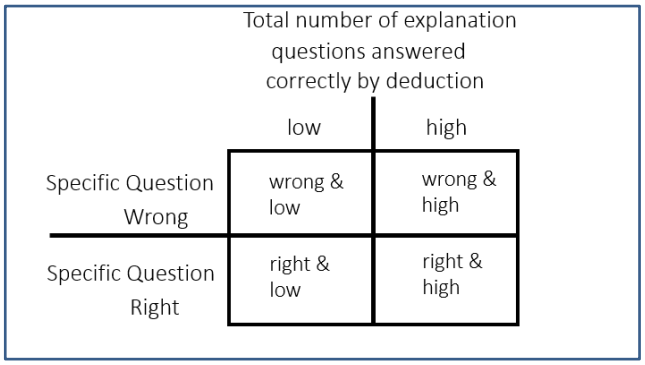

Figure 4. Options for Comparing Novices.

Recall that, earlier in this paper, I advocated testing all the participating novices on one or more specific code writing tasks. It would also be interesting to compare the eye movement data of novices who split into those who scored high on the writing tasks and those who did not.

4.4 Have the Novice Think Aloud

Donna Teague's thesis [10] is full of interesting insights elicited by having students think aloud [12] as they solved programming problems, including explanation questions. I think it would be interesting to link eye movement data with what novices were saying as they worked on an explanation question.

4.4 Cast Your Net Wide

Prior to collecting eye movement data, I suggest trialling potential explanation questions on many novices, then collecting eye movement data just for those explanation questions on which novices gave interesting or surprising answers. Some interesting and/or surprising explanation questions have already been identified in the published literature.

5. SOME EXPLANATION QUESTIONS

The explanation questions presented below elicited interesting and/or surprising results from a set of twelve explanation questions studied by Pelchen and Lister [8]. These explanation questions were given to a class of several hundred students as part of an exam at the end of their first semester of learning to program. The code of all questions was Java, but the questions can easily be translated into many other languages.

5.1 Sum All the (Positive) Values in an Array

The code for two explanation questions is shown in Figure 5. In one of the explanation questions, all the code in Figure 5 is used, and a suitable explanation for that code is "it sums all the positive numbers in the array". I shall refer to that question as "PosSum". In the other explanation question, the shaded line of code in Figure 5 is omitted (i.e., the line beginning `if`) and a suitable explanation for that code is "it sums all the numbers in the array". I shall refer to that question as "Sum".

```
int z = 0;
for (int i=0 ; i<x.length ; ++i ) {
   if ( x[i] > 0 )
      z = z + x[i];
}
System.out.println(z);
```

Figure 5: The Code for Question "PosSum", which sums all Positive Values in an Array (i.e., including the shaded line) and Question "Sum", which Sums all Values in an Array (i.e., without the shaded line).

The PosSum question was first used by Murphy et al. [7]. Those authors reported a surprising feature of their novices' answers:

> *... a common mistake was to respond that the code summed all the elements of the array. In making that mistake, students ignored the if statement within the loop – to do so is an egregious error.*

Pelchen and Lister found that some of their students made that same "egregious" error. Pelchen and Lister asked their students to explain both the PosSum code and also the Sum version. For the Sum version, 83% of their students answered correctly. However, only 70% of all the students answered PosSum correctly. Among the students who account for that 13% difference, most answered that PosSum summed all the values in the array.

If a student provides a correct answer for Sum and the same answer again for PosSum, then the student is certainly not taking into account the one line of code that differs between Sum and PosSum. But why would such a student ignore that single line of code? Eventually, after much pondering, I realized there was a flaw in my thinking – could it be that such a student ignored (or at least paid little attention to) more than just that one line of code? I was then led to the following conjecture: such a student pays most attention to just two lines of code, which occur in both Sum and PosSum. One of the lines is the print statement, which establishes that it is the value in `z` that is outputted. The other is the line in the loop body where `z` is updated. That conjecture leads me to offer the following question for an eye movement study:

- **Research Question:** Among students who answer question Sum correctly, are there eye movement differences between those students who answer PosSum correctly and students who answered incorrectly that PosSum sums all values?

If it proves to be the case that some students are correctly answering Sum while paying little attention to most of the code, then Sum is a less valid question than PosSum for establishing that a student truly understands the code.

When collecting eye movement data for this research question, I offer two minor recommendations on method: (1) Do not ask Sum and PosSum consecutively, and (2) perhaps do not use the variable names `x` and `z` in both questions.

5.2 Counts Identical Values in Two Arrays

Figure 6 shows code where a correct explanation for that code is something like "it counts the numbers of values that occur in both arrays". A note similar to that at the top of Figure 6 was provided to students.

Pelchen and Lister reported that 54% of students answered this question correctly. That approximately half the students answered this question correctly was a surprise to me, as this was the longest piece of code presented to the students, and one of the most difficult algorithms. On reflection, however, it is my conjecture that students probably only had to carefully read the lines in Figure 6 that are shaded. Given the use of a variable named `count` and the outputting of that variable's value at the end of the code, a student could easily infer that the purpose of the code was to count something. In addition, the variable `count` is incremented at only one place in the code, immediately after the `if` condition that tests whether an element in one array is equal to an element in the other array. This conjecture leads me to the following research question:

- **Research Question:** Among students who answer this question correctly, how much of the code do they actually look at closely?

This is not a clear research question; I leave it to the reader to quantify "look at closely" in terms of eye movement data.

When I wrote this explanation question, I elected to use a meaningful variable name, `count`. Some readers might prefer to instead use a meaningless variable name. Also, I elected to run the scan of the arrays "backwards", from high index values to low index values. Doing so is probably unnecessary and readers might prefer to rewrite the code to scan in the conventional direction.

If it proves to be the case that some students are correctly answering this question while paying little attention to most of the code, then this is not an explanation question where a correct answer establishes with confidence that a student truly understands the code. However, there are two ways of modifying this code that

Note: In the code below, `x1` and `x2` are arrays of any length, the elements in each array are sorted in ascending order (i.e., from smallest to largest), and no number occurs more than once in the same array.

```
int i1 = x1.length-1;
int i2 = x2.length-1;

int count = 0;

while ((i1 >= 0 ) && (i2 >= 0 ))
{
    if ( x1[i1] == x2[i2] )
    {
      ++count;

      --i1;
      --i2;
    }
    else
    if (x1[i1] < x2[i2])
    {
      --i2;
    }
    else
    {
      --i1;
    }
}

System.out.println(count);
```

(move)

(copy)

Figure 6: Code that Counts Identical Values in Two Sorted Arrays. Arrows indicate possible changes for harder questions.

might lead to a better explanation question. One modification would be to move the increment of `count` from its current location to the body of the second `if` condition. In Figure 6, this movement of the increment of `count` is indicated by the upper of the two arrows. The code would then count the number of values that occur in one of the arrays but not in the other array. The other possibility is to modify the code even further, adding a second increment of `count` which is executed when none of the `if` conditions are true (i.e., it is added to the final `else` block). In Figure 6, this copying of the increment of `count` is indicated by the lower arrow. With that modification, the code then counts the number of values that occur in array `x1` but not in `x2`, or in `x2` but not in `x1`. As either of those modified forms of the code are, I suspect, more difficult than the original code, my intuition is to retain the meaningful variable name, `count` and run the array scans in the conventional "forward" direction.

5.3 Prints the Largest of Three Values

Figure 7 shows code for a question where a suitable explanation for the code is "it prints the largest value stored in the three variables `a`, `b` and `c`". Pelchen and Lister reported that 78% of their students answered this question correctly, making it the easiest question of the twelve explanation questions they presented to students.

```
if ( a < b) {
   if ( b < c)
      System.out.println(c);
   else
      System.out.println(b);
}
else {
   if ( a < c)
      System.out.println(c);
   else
      System.out.println(a);
}
```

Figure 7: Code that Prints the Largest of Three Values.

What makes this an interesting question for an eye movement study is that Thomas Pelchen (private communication) noticed that some students provided a strange incorrect answer; an answer like, "It prints out the largest value stored in the four variables `a`, `b`, `c` and `d`" – but the code does not have a fourth variable called `d`!

My conjecture is that the students who gave this strange answer did not read all the code. Instead, after reading the first 3 to 5 lines of code, they correctly guessed that the code was finding the maximum value, but in their subsequent quick scan of the rest of the code, they counted a total of four output statements and thus incorrectly inferred that the code contained four variables. This conjecture leads me to the following research question:

- **Research Question:** Are there eye movement differences between the students who only mention variables `a`, `b` and `c` in a correct answer, and students who mention the phantom variable `d`?

In the past, when I have presented the code in Figure 7 at seminars and conferences, it has sometimes been put to me that the code shown in Figure 8 is a better way to code how the largest of the three values can be found. Even if that is true, it doesn't alter the point that some students have trouble correctly explaining the code in Figure 7. It might be interesting to conduct a study where students have to explain the code in both Figure 7 and Figure 8. If anyone does such a study, I recommend that the code from one of these figures is not presented to a novice immediately after the other piece of code is presented.

```
if ( x < y) {
   t = y
else
   t = x

if ( t < z) {
   t = z

System.out.println(t);
```

Figure 8: Alternate Code for Printing the Largest of Three Values.

5.4 Searching for a Value in an Array

Figure 9 shows code for a question where a minimally suitable explanation for the code is "It searches the array for the value in variable `x`". The interesting aspect of this question is that students can provide answers satisfying several criteria:

- The code searches the array for the value in `x`. (80%) An answer like this is the minimally acceptable answer.
- Returns the position of the search value in `x`. (70%)
- Returns -1 if the search value is not found. (54%)
- Returns the last position if the search value occurs more than once. (24%)

In the above list, each percentage in parentheses is the percentage of students who satisfied that criterion in the Pelchen and Lister study.

```
int q (int data[], int x ) {

int z = -1;

for (int i=0; i < data.length; i++ )
{
    if( data[i] == x )
       z = i;
}
return z;
```

Figure 9: Code that Searches for a Value in an Array.

Given that student answers can satisfy differing criteria, I am led to offer the following research question:

- **Research Question:** Do students who provide answers satisfying differing criteria have different eye movements than the students who just give the minimal answer?

When data is collected, I suspect there will only be minor differences in eye movements. Perhaps the students who provide a minimally acceptable answer will attend less to the first three lines of code and focus instead on the body of the `for` loop and the `return` statement. But for the students who provide more elaborate answers, I suspect the differences between them are less in the movement of their eyes and more in the movement of their "mind's eye" (i.e., their thinking). This conjecture by me about what may or may not be seen in eye movement data illustrates a more general issue with eye movement data – it only tells us what the student looked at; not what the student was thinking while they looked. The most definite result eye movement data can possibly ever give is what a student did NOT look at, and therefore could not possibly have thought about.

5.5 Checking If an Array is Sorted

Recall the explanation question from Figure 1. A minimally correct explanation for that code is "It checks if the array is sorted". Pelchen and Lister reported that 59% of their students answered this question correctly.

This code is worthy of an eye movement study because of its place in the literature on code explanation questions. It was the first such question studied [2, 14] and it is probably the most often studied code explanation question.

As with the previous explanation question, I think the issue here is whether the differences between students who get this question right or wrong will be differences in the eye movements or differences in the movement of the mind's eye. However, perhaps novices who get this question right will spend more time looking at the line commencing with `if`.

6. CONCLUSION

An exam paper is like a Turing test – we ask the students some questions and then evaluate whether any genuine thought lies behind the answers. However, the data from both Turing tests and programming exams can be ambiguous. In some of the code explanation questions presented in this paper, we have seen how a novice might correctly guess what the code does without having a genuine understanding of the code. Asking students to think-aloud as they read and write code has elicited some insight into how novices understand programs and program-writing, but much about the novice programmer remains a mystery. Eye movement studies have the potential to throw further light on the enigma that is the novice programmer.

I hope the insights from my 20 years of research are useful to those who in the future will collect and study the eye movement data of novice programmers. I look forward to reading the results from those future eye movement studies, especially for the explanation questions I have included in this paper.

ACKNOWLEDGMENTS

I thank Teresa Busjahn for her comments on a draft of this paper (but all mistakes are mine). I thank my many collaborators who, over the years, studied with me code tracing and code explanation. I especially thank Tony Clear and Donna Kingsbury (formerly Donna Teague). I suspect it was John Hamer who suggested the code used in the first explanation question and if that suspicion is right, I thank him for his intuition. I also thank my partner, Ilona Box, for her support, her patience, her ideas, and her companionship. Finally, I thank Connie, Sam and Lulu, for their love and their patience, if not their understanding.

REFERENCES

[1] Raymond Lister, Elizabeth S. Adams, Sue C. Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate E Sanders, Otto Seppälä, Beth Simon, Lynda A Thomas (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGSCE Bulletin*, 36(4), 119-150. https://dl.acm.org/doi/10.1145/1041624.1041673

[2] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, Christine Prasad (2006) Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin* 38 (3), 118-122. https://dl.acm.org/doi/10.1145/1140124.1140157

[3] Raymond Lister (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Thirteenth Australasian Computer Education Conference, Perth*. https://dl.acm.org/doi/10.5555/2459936.2459938

[4] Raymond Lister (2016) Toward a Developmental Epistemology of Computer Programming. *Keynote paper/address at Workshop in Primary and Secondary Computing Education* (*WiPSCE*), *Münster, Germany*, 13 - 15 Oct 2016. pp. 5-16. https://doi.org/10.1145/2978249.2978251

[5] Raymond Lister (2020) On the cognitive development of the novice programmer: and the development of a computing education researcher. *Keynote paper/address at the 9th Computer Science Education Research Conference (CSERC '20)*. pp. 1–15. https://doi.org/10.1145/3442481.3442498

[6] Mike Lopez, Jacqueline L. Whalley, Phil Robbins, Raymond Lister (2008) Relationships between reading, tracing and writing skills in introductory programming. *Fourth International Workshop on Computing Education Research* (*Sydney, Australia, September 6 - 7*). ICER '08. ACM, New York, NY, 101-112. https://dl.acm.org/doi/10.1145/1404520.1404531

[7] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. (2012). *Ability to 'explain in plain english' linked to proficiency in computer-based programming.* In Proceedings of the ninth annual international conference on computing education research (ICER '12). ACM, New York, NY, USA, 111-118. http://doi.acm.org/10.1145/2361276.2361299

[8] Thomas Pelchen and Raymond Lister. 2019. *On the Frequency of Words Used in Answers to Explain in Plain English Questions by Novice Programmers*. In Proceedings of the Twenty-First Australasian Computing Education Conference (ACE '19). Association for Computing Machinery, New York, NY, USA, 11–20. DOI:https://doi.org/10.1145/3286960.3286962

[9] Juha Sorva. 2012. Visual program simulation in introductory programming education. (Doctoral dissertation). Aalto University, Espoo, Finland. ISBN (printed) 978-952-60-4625-9. https://aaltodoc.aalto.fi/handle/123456789/3534

[10] Donna Teague. 2015. *Neo-Piagetian Theory and the Novice Programmer*. Ph.D Thesis. Queensland University of Technology. http://eprints.qut.edu.au/86690/1/Donna_Teague_Thesis.pdf

[11] Lynda Thomas, Mark Ratcliffe, Benjy Thomasson. 2004 Scaffolding with object diagrams in first year programming classes: some unexpected results. *SIGCSE Bull*. 36, 1, pp 250-254. http://doi.acm.org/10.1145/1028174.971390

[12] Maarten W. van Someren, Yvonne F. Barnard, and Jacobijn A.C. Sandberg (1994)**.** *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press.

[13] Anne Venables, Grace Tan, Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *International Computing Education Research Workshop (ICER), Berkeley, California, August 10-11*, 117-128. http://doi.acm.org/10.1145/1584322.1584336

[14] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins Phil, P K Ajith Kumar, Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. *Proceedings of the 8th Australasian Conference on Computing Education*, 243–252. https://dl.acm.org/doi/pdf/10.5555/1151869.1151901

[15] Wikipedia. 2022. Cargo cult programming. https://en.wikipedia.org/wiki/Cargo_cult_programming [Accessed March 2022].

[16] Wikipedia. 2022. Programming by permutation. https://en.wikipedia.org/wiki/Programming_by_permutation [Accessed March 2022].

[17] Wikipedia. 2022. Shotgun debugging. https://en.wikipedia.org/wiki/Shotgun_debugging [Accessed March 2022].

[18] Wikipedia. 2022. Voodoo programming. https://en.wikipedia.org/wiki/Voodoo_programming [Accessed March 2022].