

# Blinded by their Plight: Tracing and the Preoperational Programmer

Donna Teague

*Queensland University of Technology*  
*Australia*  
*d.teague@qut.edu.au*

Raymond Lister

*University of Technology, Sydney*  
*Australia*  
*raymond.lister@uts.edu.au*

Keywords: POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-V.B. protocol analysis, POP-VI.E. computer science education research

## Abstract

In this paper, we present evidence that some novice programmers have the ability to hand execute (“trace”) small pieces of code and yet are not able to explain what that code does. That evidence is consistent with neo-Piagetian stage theory of programming. Novices who cannot trace code are working at the first stage, the sensorimotor stage. Novices who are working at the preoperational stage, the second stage, can trace code but do not yet have a well-developed ability to reason about the code’s purpose, other than by induction from input/output pairs. The third stage, the concrete operational stage, is the first stage where novices can reliably reason about code. We present data from think aloud sessions that contrast the behaviour of preoperational and concrete students while they attempt to reason about code.

## 1. Neo-Piagetian Stages of Development

Lister (2011) proposed that we could describe novice programmers’ behaviour using neo-Piagetian stage theory. This theory is based on the premise that there are consecutive, cumulative stages through which we develop increasingly more mature abstract reasoning and expertise in a domain.

### 1.1 Sensorimotor Stage

Sensorimotor is the first, and least mature, stage of development. It is at this stage that misconceptions about basic programming concepts most influence the novice’s behaviour, like those misconceptions described by du Boulay (1989). A sensorimotor novice programmer has as yet minimal language skills in the domain and is still learning to recognise syntax and distinguish between the various elements of code. At this stage the novice requires considerable effort to trace code (i.e., hand execute), and only occasionally do they manage to do so accurately.

### 1.2 Preoperational Stage

At the next more mature stage, the preoperational novice has made headway into mastering basic programming concepts, with most misconceptions having now been rectified. This makes it possible for them to more consistently trace code accurately. However, the preoperational novice is heavily reliant on the use of specific values to trace, understand and write code. Preoperational novices are not yet able to perform abstract reasoning about a chunk of code, as their focus is quite narrow: limited to simply a single statement or expression at a time. They struggle to recognise the relationship between two or more statements.

### 1.3 Concrete Operational Stage

By the time a novice is at the concrete operational stage, their focus shifts from individual statements to small chunks of code which allows them to consider the overall purpose of code. Their ability to reason at a more abstract level allows them to understand short pieces of code simply by reading that code. When the concrete operational novice does trace, they can do so in an abstract manner rather than being reliant on the use of specific variable values. One of the defining characteristics of the

concrete operational novice is the ability to perform transitive inference: comparing two objects via an intermediary object. For example, if  $A > B$  and  $B > C$  then, by transitive inference,  $A > C$ .

#### 1.4 Piaget -v- Neo-Piaget

Whereas Piaget himself focussed on the cognitive development of children, neo-Piagetian theory is concerned with cognitive development of people of any age, learning any new task. A person can thus concurrently exhibit characteristics from different stages in different knowledge domains. Using a methodology based on Piaget's theory of genetic epistemology, da Rosa (2007) witnessed in her research participants the transition of reasoning about relationships towards the construction of new recursive concepts. According to neo-Piagetian theory, time taken by individuals to transition through the stages varies, but there are conflicting theories about the nature of those transitions which we will discuss in the next section.

#### 1.5 Staircase Model -v- Overlapping Wave Model

We previously alluded to conflicting theories about the nature of the transitions between neo-Piagetian stages. Although theorists agree that the neo-Piagetian stages are consecutive and cumulative, one view is that the stages are discrete, much like a stair-case model. However, there is a growing body of evidence suggesting that progress through the stages may not be so straightforward. How, for example, does one make the quantum leap from one stage to the next? An alternative to the stair-case model is that the transition through the stages can be seen as overlapping waves: where a person exhibits characteristics from two or more stages as they develop skills in the domain (Siegler, 1996; Boom, 2004; Feldman, 2004). In this overlapping waves model, characteristics of the earliest stage dominate behaviours initially, but as cognitive progress is made there is an increase in use of the next more mature level of reasoning and a decrease in the less mature. In this way, there is concurrent use of multiple stages of reasoning. This model is depicted in Figure 1. As will be apparent later in this paper, some novice programmers' behaviour that we have observed fits this overlapping waves model.

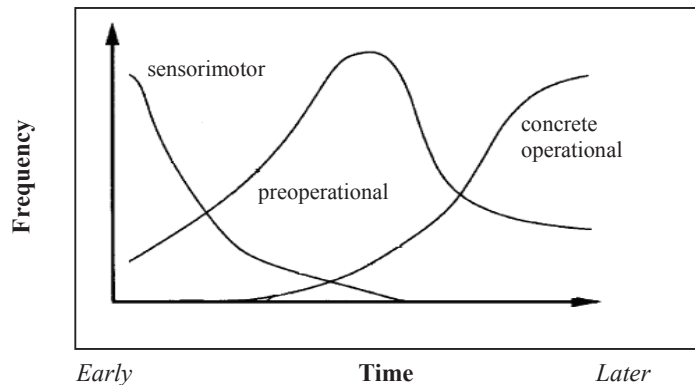


Figure 1 Overlapping Waves Model

## 2. Methodology

Previous studies have found evidence that novices find explaining code harder than tracing code (Lister, Simon, Thompson, Whalley, & Prasad, 2006; Whalley et al., 2006; Lister, Fidge, & Teague, 2009; Simon, Lopez, Sutton, & Clear, 2009). Philpott, Robbins and Whalley (2007) found that students who could not accurately trace were not able to explain similar code.

But if a novice has the skills to accurately *trace* a piece of code, shouldn't they then have an adequate understanding of it to be able to explain the purpose of that same piece of code? In this paper, we gathered empirical data to help us answer this question. We gave students some code and asked them to both *trace* and *explain* its purpose.

At most institutions, Explain in plain English (EPE) questions are not as familiar to most programming students as tracing and writing tasks. At our institution, however, students encountered EPE questions in their lectures and previous tests and therefore were familiar with what type of

answer was expected of them. They had not necessarily both traced and explained the same piece of code before. We asked students to do that in this study, first to establish that they understood the semantics of the code by being able to trace it, and second, if they could indeed trace it, to determine if they were also able to explain its purpose.

## 2.1 In-Class Testing

We tested introductory programming students with trace and explain tasks at our university during their lecture in four different (13 week) semesters. The students involved in these in-class tests had already completed one semester of programming. Students were asked to complete the tests individually, as if they were sitting an exam, but the tests did not contribute to their final grades.

In the sixth week of each of the four semesters, we gave the students the programming tasks shown below in Figure 2. (Note that the line numbers next to the code in Figure 2 were inserted by the authors of this paper for the readers' benefit, and were not part of the exercise given to the students.) Sample answers are provided in the shaded areas of the figure. The concepts that these tasks use (selection and output) were covered in week four of their first unit of study, so in effect, students had exposure to these concepts for 15 teaching weeks. In other words, the programming concepts in the tasks were quite familiar to them.

Consider the following block of code, where variables <i>a</i> , <i>b</i> and <i>c</i> each store integer values:	
<pre> 1      if (a &gt; b) { 2          if (b &gt; c) { 3              Console.WriteLine(c); 4          } else { 5              Console.WriteLine(b); 6          } 7      } else if (a &gt; c) { 8          Console.WriteLine(c); 9      } else { 10         Console.WriteLine(a); 11     } </pre>	
(a) In relation to the above block of code, which one of the following values for the variables will cause the value in variable <i>b</i> to be printed?	
<p>(i) <i>a</i> = 1; <i>b</i> = 2; <i>c</i> = 3;</p> <p>(iii) <i>a</i> = 2; <i>b</i> = 1; <i>c</i> = 3;</p>	<p>(ii) <i>a</i> = 1; <i>b</i> = 3; <i>c</i> = 2;</p> <p>(iv) <i>a</i> = 3; <i>b</i> = 2; <i>c</i> = 1;</p>
Correct answer:	(iii)
(b) In one sentence that you should write in the box below, describe the purpose of the above code (i.e. the <i>if/else if/else</i> block). Do <b>NOT</b> give a line-by-line description of what the code does. Instead, tell us the purpose of the code:	
Sample answer:	<i>To print the smallest of the three given values.</i>

Figure 2 - Trace and Explain Tasks

Although informally invigilated, our in-class testing was not conducted under formal exam conditions and students may have been less motivated to complete the tests than if those tests had contributed to their grades. On the other hand, our students also had less motivation to plagiarise.

## 2.2 Think Aloud Sessions

An issue with any type of written exam is that test scripts (i.e., the papers that the students hand in) are sometimes not an accurate indication of students' ability at all, and certainly rarely give any insight into the process they used to arrive at an answer (Teague et al., 2012). We wanted evidence of how students traced and reasoned about code. Artefacts from think aloud sessions are potentially a much richer source of data which describe the students' *process* of solving tasks (Ericsson & Simon,

1993; Atman & Bursic, 1998). So to complement any quantitative findings from our in-class tests, we also ran a series of think aloud sessions with volunteer students from our first two introductory programming classes and asked them to complete an exercise similar to that shown in Figure 2. Note that our think aloud students came from different cohorts using different programming languages, but essentially there were only syntactic differences in the code. The exercise was presented to them in the same manner as the in-class test, except the exercise was printed on special dot paper on which they wrote their answers with a SmartPen (LiveScribe, 2014). This allowed us to see what students wrote and record what they said as they did so.

### 3. Results

#### 3.1 In-Class Testing Results

Table 1 shows the performance of students on the tasks in Figure 2, from four different cohorts, in four different semesters (each cohort is a row in the table). The last row of that table combines the four cohorts.

A great proportion of the students we tested over the four semesters were able to answer the tracing question correctly (see Col. 3 of Table 1). A much smaller percentage could actually explain the code (see Col. 5 of Table 1). A total of 29% of the students *could* trace the code and therefore had a working knowledge of the programming concepts involved, but *could not* explain what that code did (see Col. 2 of Table 1).

Considering students were working with the same code for both tasks, these results seem to be surprising. Why were so many students unable to explain that code when they could trace that code?

n	Col. 1 Can trace (a) and can explain (b)	Col. 2 <b>Can trace (a) but cannot explain (b)</b>	Col. 3 Can trace	Col. 4 Cannot trace (a) but can explain (b)	Col. 5 Can explain	Col. 6 Can neither trace (a) nor explain (b)
51	31 (61%)	<b>10 (20%)</b>	41 (80%)	1 (2%)	32 (63%)	9 (18%)
113	40 (35%)	<b>31 (27%)</b>	71 (63%)	0 (0%)	40 (35%)	42 (37%)
53	27 (51%)	<b>21 (40%)</b>	48 (91%)	1 (2%)	28 (53%)	5 (9%)
86	51 (59%)	<b>26 (30%)</b>	77 (90%)	5 (6%)	56 (65%)	4 (5%)
303	149 (49%)	<b>88 (29%)</b>	237 (78%)	7 (2%)	156 (51%)	60 (20%)

Table 1 Comparison of students' performance on the trace and/or explain tasks in Figure 2

#### 3.2 Think Aloud Sessions Results

As we have seen, a significant number of students in our in-class tests were not able to explain the code even though they could trace it. However, only two of the students who took part in think aloud sessions were able to trace the code but were not able to explain the code. Although two students is much too small a sample size from which to draw conclusions or generalise (about why students are not able to explain the purpose of code), an analysis of these two students' *process* of completing the tasks is insightful. It gives us evidence that their ability to do one task and not the other can be explained by neo-Piagetian theory.

For anonymity, our think aloud students chose aliases, by which we will refer to them. The think aloud sessions with four students are summarised in Table 2. Later in the paper, we will discuss in detail the difficulties encountered by two of those students, Michael and Charlotte, as they completed the tasks in Figure 2. But first, by way of comparison, we introduce the other two students, Lance and Briandan, who completed the exercise without difficulty, and did so in ways we had originally anticipated all of our students would complete it.

Alias	Weeks of prior programming instruction	Time taken to complete exercise (min:seconds)	Dominant neo-Piagetian stage demonstrated by behaviours
Lance	10	2:14	Concrete
Briandan	26	3:06	Preop/Concrete
Michael	6	7:26	Preoperational
Charlotte	6	8:03	Sensorimotor/Preoperational

*Table 2 Summary of Think Aloud Sessions, with Subjects in Order of Reasoning Sophistication*

Our detailed excerpts which follow use a format similar to that used previously in qualitative studies (Lewis, 2012; Teague & Lister, 2014a), where the interview data is presented separate to its analysis, so that the reader may more easily follow the think aloud session.

In this paper, pauses in speech are marked “...”, as placeholders for dialog we have not included because we deemed that the excluded dialog added nothing to the context of the think aloud session. Utterances are italicised and where we have added our own annotations for clarification, these appear in square brackets in non-italicised text.

### 3.2.3 Lance

#### *Summary*

Lance completed a Python version of the exercise in Figure 2 without any fuss, in little more than 2 minutes. He did not trace with specific values. While reading the code in Figure 2 he spontaneously determined the purpose of the code.

#### *Data*

After very quickly reading the code almost in its entirety, Lance made the comment:

Lance: *Ah that's a bit of a mind warp.*

He made a mark in the code (line 5 in Figure 2) indicating the part of the code that needed to execute.

Lance: *... so basically to get there [line 5] we need a to be greater than b and we need b to be greater than c ... oh no we need b to be ... less than c ... so we need b to be the smallest number*

To verify his thinking, Lance then traced the code with the set of values in option (iii)

Lance: *a ... is greater than ... b ... yes ... b is ... greater than c ... no ... so it doesn't print c ... and then it goes to the else statement print b so ... yep so (iii)*

As part of answering part (a) in Figure 2, Lance had already explained the purpose of the code, so he was able to write his answer to part (b) without hesitation.

#### *Analysis*

Lance's comment about the code being “*a bit of a mind warp*” makes us believe that on first reading, he had not formed a clear understanding of the code. However, he then determined which conditions must be met in order for the required output statement to print. We refer to this as a “backward” trace. The ease with which he volunteered an explanation of the purpose of the code in part (b) of Figure 2 indicates that he had already processed a great deal of the code's semantics while completing part (a).

Lance had no real need to trace the code with the specific values in the options, as his abstract trace of the code and conclusion that “*we need b to be the smallest number*” was sufficient to identify the correct option. However, he traced the code using the values from option (iii) to confirm the answer at which he had arrived. He did not at any stage refer to any of the code after line 6 that is, the `else` branches of the first `if` block.

Lance formed a coherent understanding of the “big picture” as a by-product of his trace. It was his grasp of the concept of *transitive inference* (comparing two things via an intermediary) that allowed him to quickly determine the code’s purpose, that is, that if  $a > b$  and  $b < c$  then  $b$  is the smallest. Transitive inference is a defining quality of the concrete operational stage.

### 3.2.1 Briandan

#### *Summary*

Briandan completed a Java version of the task in part (a) of Figure 2 by doing a “backward” trace, much like Lance had done. However, she was a little more reliant on specific values than Lance. Briandan eliminated two options based on the first condition not being met, then traced the code with the values given in the remaining two options to find the correct option. For the EPE task (part (b) in Figure 2) she did little else other than to re-read the given code, consider its purpose and then correctly describe that purpose. Whereas Lance had already spontaneously formed a clear understanding of the code as part of his trace, Briandan had not.

#### *Data*

Briandan read each section of the code, sometimes articulating a summary rather than each token of the syntax. For example, she said “*print*” to summarise the code “*system.out.println*”. (Note that as Briandan was working in Java, the “*print*” statement was different to that shown in Figure 2.)

Briandan drew a line next to the code at line 5 in Figure 2 which prints  $b$ , to indicate the line that must be executed. She then determined that the first condition at line 1 in Figure 2 (i.e.,  $a > b$ ) must be true in order for  $b$  to print, and marked that condition with a dash. She said:

Briandan: *let’s eliminate [options] if a ... greater than b ... we need a greater than ... no*

She then crossed out option (i) and proceeded to check the other three options in a similar way:

Briandan: *... is a greater than b yes ... that one [option (iii)] ... a greater than b, no [option (ii)] ... a greater than b ... mm yes [option (iv)] ... so we’ve got two options here*

By this stage she had eliminated options (i) and (ii) and placed a mark under options (iii) and (iv). However, she then changed tactic, and traced the values provided in options (iii) and (iv):

Briandan: *Now if b greater than c b ... greater than c ... no it’s not ... so we’re going to go to the else one ... so that would be possible [option (iii)]*

She then tested the final option as well in order to confirm her choice:

Briandan: *and other one [option (iv)]... yeah it would print c*

Briandan circled the correct option (iii).

To answer the EPE task in part (b) of Figure 2, Briandan read the code again then said:

Briandan: *hold on ... we printed the ... smallest number ... so assuming ... this didn’t work right [i.e., the condition ( $b > c$ ) at line 2 in Figure 2 failed]... if c greater than b its going to go up here [line 5 in Figure 2]... so printing the smallest number*

#### *Analysis*

That Briandan substituted some meaning for expressions as she read the code is evidence that she was processing the code, rather than simply reading it (i.e., when she said “*print*” in lieu of the code “*system.out.println*”).

Briandan’s initial approach to doing the tracing task was a short-cut elimination of two options based on the conditions that must be met. She did not test each answer option in a linear fashion. However, having eliminated two options by this approach, she then changed to tracing the remaining two options to determine their outcome. She did not make a transitive inference. Like Lance, Briandan found no need to refer to the code after line 6 in Figure 2.

In part (b), and unlike Lance, Briandan had to think further about the meaning of the code: she had not deduced the meaning of the code as she traced in part (a) in Figure 2. It hadn't occurred to her during the trace that the code would always print the smallest number. That is, she did not see that a transitive inference could be made about the variables.

### 3.2.2 Michael

#### *Summary*

Michael was given the C# exercise in Figure 2. His approach to the tracing task was different. He started with the first option and (forward) traced its values. When he noticed the  $(a > b)$  condition (at line 1 in Figure 2) was not met, he was able to eliminate both option (i) and option (ii). He then traced with the values from option (iii), and determined that  $b$ 's value would be printed. He chose option (iii) as the correct answer, and did not trace option (iv). Even though he accurately traced the code, he then had difficulty explaining what the code did autonomously. To explain the code, he required scaffolding from the interviewer.

#### *Data*

Michael read the code verbatim, including each output statement and punctuation ("*console dot writeline...*"). He then took the values in option (i) and started tracing:

Michael: *a equals to 1 b equals 2 c equals 3. 1 is more than 2 no. So this if statement would not run [i.e., the condition  $(a > b)$  at line 1 in Figure 2 would evaluate to false].*

Recognising that the `if (a > b)` block needed to execute for  $b$  to be printed, and therefore  $a$ 's value needed to be greater than  $b$ 's, he eliminated options (i) and (ii).

Michael: *so just left with the third option and fourth option*

He then looked at the next option, (iii):

Michael: *So a ... is more than b ... 2 is more than 1. You jump to next statement where 1 is more than 3 console dot writeline c. ... doesn't happen [i.e., the condition  $(b > c)$  at line 2 in Figure 2 fails] ... so we write b. ... Yeah. ... so it's (iii)*

After requiring clarification that a line by line description was not required for part (b), Michael gave his first and incorrect explanation of the code:

Michael: *Display the values of a b and c?*

Interviewer: *Would it display all of them?*

Michael: *Not all of them. It depends on what starting values they have*

Interviewer: *Under what conditions would a print?*

Michael: *from the code ... as long as ... a is the smallest number ... comparison to ...*

At this point, Michael seemed to have figured out the purpose of the code. However, when asked under what conditions would  $b$  print he attempted to answer in terms of each of the conditions in the code that would need to be met:

Michael: *b would print when um ... a is smaller, a is larger than b, plus smaller than c ... and ... but ... oh wait wait wait I take that back. Uh ... a is ah ... more than b ... but less than c ... and c is larger than b ... ah c is ... larger than b yeah... to print b*

The interviewer asked a similar question about the code printing  $c$ , to which he gave a similar, preoperational answer. Then he was asked a more general question about the code:

Interviewer: *What can you say about it printing a or b or c? Is there anything in common?*

Michael: *Yeah all depends on values of a ... ah ... I'm very confused*

The interviewer then gave Michael a set of values for  $a$   $b$  and  $c$  to trace: 10, 2, and 7 respectively.

Interviewer: *Which one will print?*

Michael: ... *um 2* ...

The interviewer then gave Michael another set: 2, 5, and 12 respectively:

Michael: ... *a ... yeah a ... I've figured it out <laugh> as long as ... it's the smallest digit it will be printed*

### **Analysis**

Unlike Briandan, Michael's inclination was to read every token of the code including output statements and punctuation verbatim ("*console dot writeline...*"). This is a remnant of the sensorimotor stage, where the language in the domain is still developing. At this stage, processing the meaning of many elements of code is necessary, and tends to be a time consuming process that requires a great deal of cognitive effort.

Michael traced the values in the options in a mechanical manner. He chose the first option, determined the outcome, eliminated another option based on the outcome of the first, then traced option (iii) before deciding it was the correct one.

Michael continually referred to the specific values of the variables while he traced. For example, "*I is more than 2 no*", "*a ... is more than b ... 2 is more than 1*". Reliance on specific values to trace is consistent with preoperational behaviour.

Michael did not attempt to reason about the code as he traced. Unlike Lance, he was not building an understanding of the code's purpose while he traced it. His attempt to explain the code (part (b) in Figure 2) showed he had very little ability to reason about the code's purpose which is, again, indicative of someone at the preoperational stage. His explanation relied on inductive reasoning based on various input/output pairs. He itemised which conditions in the code needed to be met before the line to print *b* would be executed. As with any preoperational novice, Michael was preoccupied with evaluating individual statements rather than developing a more abstract "big picture".

It was only after several prompts by the interviewer, which lead Michael through additional traces with specific values, that he understood the purpose of the code. Until what appeared to be a "light bulb moment", Michael had used specific values when asked which variable would print: "... *um 2 ...*", but after his enlightenment, his responses became abstract: "... *a ... yeah a ...*". It was as if he had only just come to realise that the code's outcome, when expressed abstractly, was invariant.

### 3.2.3 Charlotte

#### **Summary**

Charlotte was given the C# exercise in Figure 2. Her method of tracing the code involved substituting specific values for each of the variables. She rewrote the code in specific rather than abstract terms. After an initial self-corrected error, she determined that the first condition (i.e.,  $(a > b)$ ) at line 1 in Figure 2) must be met in order for the correct output statement to be executed, and then eliminated options (i) and (ii). She rewrote the code using values from option (iii), decided it was correct, but also checked (iv) before eliminating it. Charlotte's attempt to explain the purpose of the code was also heavily reliant on the use of specific values.

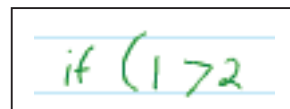
#### **Data**

Charlotte read the code including "*curly brace ... writeline ...*", and circled the "b" in the output statement (at line 5 in Figure 2) as a "note to self". She then proceeded to rewrite the code with specific values (from option (i)) substituted for the variables. She said:

Charlotte: *I'm going to write them out*

Charlotte then wrote what can be seen in Figure 3. Immediately she made an incorrect conclusion about option (i):

Charlotte: *so this one [option (i)] could be it*



The image shows a handwritten code snippet on a white background. The text is "if (1 > 2)". The number "1" is circled in green. The rest of the code is written in blue ink.

Figure 3 Charlotte's trace of option (i)



She said of option (iii):

Charlotte: *if 2 is greater than 1 ... this won't run so (iii) is already not an option*

At this point Charlotte paused (for 32 seconds) and silently read through the problem again. She then self-corrected:

Charlotte: *so a has to be greater than b for b to print*

She then eliminated options (i) and (ii) and marked option (iv) as a possibility. She traced option (iii) again by substituting specific values for the variables and wrote what appears in Figure 4.

Figure 4 Charlotte's trace of option (iii)

Charlotte: *if 1 is greater than 3, which it's not, write c ... else ... b, so it's (iii)*

After deciding this, she also checked (iv) “just to be sure”:

Charlotte: *3 is greater than 2 which it is, if 2 is greater than 1 ... c ... no its (iii)*

When Charlotte began part (b) of Figure 2, her initial “gut instinct” (as she described it) was incorrect:

Charlotte: *showing which number is ... I guess the biggest?*

However, she was not convinced that this was correct:

Charlotte: *Ok .. don't be lazy. 'cause I don't want to go through it all again*

But she did go through it again. This time, instead of rewriting the code with values, she wrote the specific values for option (iii) above each of the variables in the given code (see Figure 5).

Figure 5 Charlotte's second trace of option (iii)

Then she asked herself:

Charlotte: *... what's so special about c?*

After a short time she concluded:

Charlotte: *In this case [referring to the line of code to output c, line 3 of Figure 2], c would be the smallest number ... in this case b [referring to line of code to output b, line 5 of Figure 2] would be the smallest number ... in this case c [referring to line of code to output c, line 3 of Figure 2] ... yeah, it's to find out what the smallest number is.*

### Analysis

Like Michael, Charlotte's inclination was to read every token of the code including punctuation (“curly brace”), which suggests that she too is manifesting remnants of sensorimotor habits.

However, some of her behaviour is clearly preoperational. She successfully traced the code, but her reliance on specific values caused her to rewrite the code substituting a specific value for each of the variables. So rather than trying to reason about abstract code, she rewrote it in a language she understood: specific values. At other times when she was tracing or verifying the correctness of her answer, she wrote values above each of the variables.

Charlotte's initial attempt at reasoning about the code's purpose (i.e., that it finds the biggest) was intuitive, and surprisingly inaccurate given that she had previously concluded that “... a has to be greater than b for b to print”. As novices at the preoperational stage attempt to reason about code, they tend to make guesses based on intuition, and those intuitions can be inconsistent.

After suspecting she was wrong, Charlotte actually considered *not* retracing. She thought better of it and admonished herself for being lazy. She then traced the code correctly. A sensorimotor novice finds any tracing task to be non-trivial and for that reason is reluctant to do so more than the minimum necessary.

In answer to part (b) of the task shown in Figure 2, it was only after, again, making extensive use of specific values and finding a pattern via inductive reasoning that she was able to make a conclusion about the purpose of the code.

Charlotte is manifesting behaviours of both the sensorimotor and preoperational stages. This behaviour fits with the overlapping waves theory as described in the previous section where her sensorimotor behaviours, although diminishing and no longer dominant, are still evident as she starts to reason at the preoperational stage.

#### 4. Discussion

Preoperational novices are heavily reliant on specific values. They talk about code in terms of specific values and trace with specific values, to the extent of replacing variables with values as they trace code like Michael did: “*a ... is more than b ... 2 is more than 1. You jump to next statement where 1 is more than 3*”. Similarly, Charlotte wrote “*if 1 > 2*”. Novices at an early phase of the preoperational stage are keenly focused on using the knowledge accumulated in the sensorimotor stage (i.e., the semantics of programming constructs) to mechanically trace code. The ability to trace in abstract terms, like Lance did, is usually beyond the preoperational novice. Michael and Charlotte are working mostly at this preoperational level.

Also beyond the capacity of the preoperational novice is the ability to reason about the purpose of the code. Preoperational novices are preoccupied with the detail of a tracing task. They have developed the ability to determine the functional outcome of each line of code and trace to completion. However the mental effort of doing so exhausts them, which obscures from them the abstract purpose of the code. They are in effect, tracing blind.

There is a stark difference between the concrete operational behaviour of Lance and the preoperational behaviour of Michael and Charlotte. With the help of the think aloud sessions we have come to understand that what Lance was doing when he traced the code was something that neither Charlotte nor Michael did when they traced the code. He was reasoning about the parts of the code as he read and traced the code. Briandan also showed some evidence of processing the code as she read it, by summarising complicated output sequences simply as “*print*”. The speed with which Briandan solved the EPE task, “*hold on ... we printed the ... smallest number ... so...*” is reasonable evidence that although she had not previously drawn this conclusion verbally, the process of tracing the code had provided some insight into the code’s purpose. This behaviour exhibited by both Lance and, to a lesser extent, Briandan is indicative of the concrete operational stage.

Charlotte would have been awarded full marks for her answer if it had been provided in an exam. We doubt that Michael would have completed the EPE question in an exam, as he was unable to do so without intervention in his think aloud session. We suspect that many of the students who completed the in-class test are much like Charlotte or Michael. Their correct test answers belie the difficulty they had with the task. (We speculate that this difficulty might explain why some students can trace code, yet not be able to write similar code.)

It is interesting that none of the think aloud students referred to code after line 6. We could account for this in a number of ways. First, students may have assumed that any code we supplied would be “purposeful” code, which is indeed the case. Second, we could attribute their behaviour to inductive reasoning. That is, they drew conclusions about the purpose of the code based on input and output combinations. By backwards tracing (i.e., finding which conditions needed to be met in order to print the value of variable *b*), they saw no need to investigate the latter section of code as it was of no consequence to the outcome in this particular instance. For example, even if lines 8 and 10 in the code were swapped, it would *still* print the smallest value when the smallest value was stored in *b*.

If we assume that the reasoning processes of the students in the in-class test are consistent with the reasoning processes of the think aloud students, then we can make some inferences about the in-class test results. Students who could neither trace nor explain (see Col. 6 of Table 1) are exhibiting behaviours that are consistent with the sensorimotor stage. They manifest limited ability to reason

logically and abstractly about the code's purpose. Therefore their attempt at an EPE task is most likely a guess.

Students who traced the code correctly but then could not explain it (see Col. 2 of Table 1) fall into the preoperational category (at best). As we have discussed previously in this paper, students working at the preoperational level, like Michael and Charlotte, have developed the skills to trace code but as yet do not have the ability to reason abstractly about its purpose. As part (a) of the task (see Figure 2) was a multiple choice question, some students would have simply guessed the correct answer in the in-class test. In that case, and if the guess was because those students were unable to trace the code, then they are students who are at the sensorimotor stage.

That there are students who could not trace the code, but yet were able to explain it (see Col. 4 of Table 1), is an anomaly, for neo-Piagetian theory. (Jean Piaget referred to such anomalies as *decalage*.) However, those students are a very small proportion of the students. We suspect they had an accurate idea of the code's purpose, but merely made a careless mistake on part (a). It is less likely that they guessed the correct explanation, as this is more difficult to do for a short answer question than a multiple choice question.

Students who were able to complete both the tracing and explaining tasks successfully (see Col. 1 of Table 1), like Lance and Briandan, *may* be working at the concrete operational level. However, it is difficult to make a conclusion based on their answer alone. It is the *process* that identifies concrete operational reasoning, not the final answer. Charlotte's think aloud session in particular argues this point. It is at the concrete operational level we would *like* all of our students to be working, and it is certainly where most of our teaching and learning material is aimed. However, as we can see from our results, many of our students fall short of this level of cognitive development because they are still preoperational, and are not yet capable of working at a concrete level, with abstractions.

Our results support previous findings that explaining code is more difficult than tracing code. Neo-Piagetian theory offers an explanation of why that is so.

## 5. Conclusion

There are important pedagogical implications that can be drawn from this research. Many of our students are not reasoning at the concrete operational level required of the type of programming tasks we expect them to complete. If they cannot reason about code given to them, then they are probably incapable of writing similar code. From our data, about a third of our students are reasoning at the preoperational stage, so to them we may as well be talking in a foreign language when we pitch our teaching resources at the concrete operational level. Our preoperational students require exposure to reading and tracing tasks which are constituted from a minimal number of parts and which give them the freedom to use a less abstract level of reasoning. With sufficient practice, and with a slow increase in the sophistication of the code they read and trace, these students will eventually reach the concrete operational stage.

Neo-Piagetian theory offers a coherent framework for explaining our data. Readers might argue that our empirical results are not entirely new, and we have cited several other similar findings. However, our use of a neo-Piagetian framework to explain such data is new. Our use of neo-Piagetian theory also has methodological implications. Knowing that tracing code does not require concrete operational skills, students who can trace code accurately are not necessarily capable of tasks that require abstract reasoning, such as explain in plain English tasks, and also writing code.

Neo-Piagetian theory suggests interesting problems on which to study students. In this particular paper, we have used a problem intended to study transitive inference. In other papers, we have used problems intended to study other aspects of concrete operational reasoning, such as reversibility and conservation (Teague & Lister, 2014a, 2014b).

## 4. References

Atman, C. J., & Bursic, K. M. (1998). Verbal Protocol Analysis as a Method to Document Engineering Student Design Processes. *Journal of Engineering Education*, 87(2), 121-132.

- Boom, J. (2004). Commentary on: Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology*, 22, 239-247.
- da Rosa, S. (2007). *The Learning of Recursive Algorithms from a Psychogenetic Perspective*. Paper presented at the Psychology of Programming Interest Group (PPIG) 19th Annual Workshop 2007, Joensuu, Finland.
- du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* (pp. 283-300). Hillsdale, NJ: Lawrence Erlbaum.
- Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.
- Feldman, D. H. (2004). Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology*, 22, 175-231.
- Lewis, C. M. (2012). *The importance of students' attention to program state: a case study of debugging behavior*. Paper presented at the 9th Annual International Conference on International Computing Education Research (ICER 2012), Auckland, New Zealand.
- Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth, WA.
- Lister, R., Fidge, C., & Teague, D. (2009). *Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming*. Paper presented at the ITiCSE 09: Proceedings of the 14th annual conference on Innovation and technology in computer science education, Paris.
- Lister, R., Simon, B., Thompson, E., Whalley, J., & Prasad, C. (2006). *Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy*. Paper presented at the Eleventh Annual Conference on Innovation Technology in Computer Science Education (ITiCSE'06), Bologna, Italy.
- LiveScribe. (2014). Retrieved March 17, 2014, from <https://www.smartpen.com.au/>
- Philpott, A., Robbins, P., & Whalley, J. (2007). *Assessing the Steps on the Road to Relational Thinking*. Paper presented at the 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Port Nelson, New Zealand.
- Siegler, R. S. (1996). *Emerging Minds*. Oxford: Oxford University Press.
- Simon, Lopez, M., Sutton, K., & Clear, T. (2009). *Surely We Must Learn to Read before We Learn to Write!* Paper presented at the 11th Australasian Computing Education Conference (ACE 2009), Wellington, New Zealand.
- Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A., & Lister, R. (2012). *Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming*. Paper presented at the Australasian Association for Engineering Education Conference (AAEE 2012), Melbourne.
- Teague, D., & Lister, R. (2014a). *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand.
- Teague, D., & Lister, R. (2014b). *Programming: Reading, Writing and Reversing*. Paper presented at the ITiCSE '14, Uppsala, Sweden.
- Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P., & Prasad, C. (2006). *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies*. Paper presented at the 8th Australasian Computing Education Conference, Hobart, Australia.