Mired in the Web: Vignettes from Charlotte and Other Novice Programmers

Donna Teague
Queensland University of Technology
Brisbane, QLD, Australia
d.teague@qut.edu.au

Raymond Lister and Alireza Ahadi
University of Technology, Sydney
Sydney, NSW, Australia
Raymond.Lister@uts.edu.au

Abstract

Ahadi and Lister (2013) found that many of their introductory programming students had fallen behind as early as week 3 of semester, and those students often then stayed behind. Our later work (Ahadi, Lister and Teague 2014) supported that finding, for students at another institution. In this paper, we go one step further than those earlier studies by observing a number of students as they complete programming tasks while thinking aloud. We describe the types of inconsistencies students manifest, which are often not evident on analysis of conventional written tests. We again interpret our findings using neo-Piagetian theory. We conclude with some thoughts on the pedagogical implications of our research results.

Keywords: Programming, neo-Piagetian theory, novices, assessment, think aloud.

1 Introduction

Many computing educators have noted a large variation in the ability of introductory programming students. Ahadi and Lister (2013) found significant differences in performance among their students, as early as week 3, on trivial coding problems. Furthermore, those students with lower scores on the week 3 test also tended to perform lower on test questions in subsequent weeks — that is, some students fall behind very early and then stay behind.

Ahadi et al. (2014) conducted a second study, comparing students at two different institutions. They found that tests held early in semester were good indicators of success in the final exam. In this paper, we report on a similar quantitative study, but we go further, by triangulating with qualitative think aloud data from students completing the same test questions.

2 Neo-Piagetian Theory

Lister (2011) proposed that we can describe students' development in programming in terms of neo-Piagetian theory. Other studies (Falkner, Vivian, and Falkner 2013; Teague et al. 2013; Teague and Lister 2014c) provide empirical evidence of novices manifesting neo-Piagetian stage-related characteristics as they reason about programming tasks. According to the evidence accumulated from these and related studies, the first three stages of development are characterised as follows.

At the **sensorimotor** stage, novices tend to inconsistently apply mis/conceptions about programming. Because of their fragile knowledge, these students struggle to successfully trace code, let alone reason about its purpose or write their own code.

At the next more mature level are **preoperational** students who have begun mastering the semantics, and any misconceptions that remain at this stage are at least applied consistently. Although preoperational students can accurately trace code, they are often not able to reason about its purpose other than by induction from input/output pairs (see Teague and Lister (2014b)).

It is at the **concrete** operational stage, the next more mature stage, where students have developed an ability to reason deductively about abstractions and write more complex code. This is the stage at which computing educators typically expect students to be working by the end of their first semester of learning programming, and the level at which students are traditionally assessed. However, the findings of this study, and previous studies**,** suggest that many students are not manifesting concrete operational skills even by their second semester of study (Teague et al. 2013).

Rather than making quantum leaps between these three stages, our view of development is described by the Overlapping Waves Model (Boom 2004; Feldman 2004; Siegler 1996). In that model, characteristics of an earlier stage dominate initially, but there is a gradual increase in the use of the next more mature level of reasoning and a decrease in the less mature stage. This model accounts for students manifesting characteristics of more than one stage simultaneously.

3 Method

The undergraduate introductory programming course we studied ran at the first author's institution over a 13 week semester comprised of a two hour lecture and a two hour workshop each week.

To collect the data for this study, students completed a short "in-class" test at the start of the lectures in weeks 2, 4, 7 and 9. These tests did not contribute to a student's final grade. However, most students present at the lecture did the test, as the lecture did not proceed until the test was over. The time students took to complete a test was not formally recorded, but each test took around 15 minutes. Students were under little time pressure. Immediately after each test, the lecturer would review the test and explain the correct answers.

Much of the work of the first author in recent years has involved observing approximately 40 individual student programmers, as they developed over the course of a semester. Those students completed programming tasks while thinking out loud (Ericsson and Simon 1993). In this paper we describe some of those students' attempts at the tasks that in-class test data identified as being problematic for many students. The qualitative data from the think aloud sessions help to answer some of the questions that arise from the in-class results:

What strategies do students use? (In other words, how did they get that answer?*)*;

What behaviour is evident with students who have difficulty completing programming tasks?; and

What programming misconceptions (if any) are evident? (Are incorrect answers a result of careless mistakes, misinterpretation of the question or lack of understanding the concept?)

Once we have that information, we can answer the "why" questions by interpreting the qualitative data using the neo-Piagetian framework:

Why do students get particular questions wrong?

Can a student have disparate levels of ability with two tasks which test similar programming concepts? (For example tracing, explaining and writing the same code.)

Why are some students unable to work with abstractions? (For example, why do they rely on tracing code with specific values?)

It is not possible to include all our think aloud data in this paper. We have simply selected three sessions that are representative of the broadly different types of reasoning manifested by our think aloud students.

We use aliases to obfuscate students' identity. Excerpts from the sessions with Charlotte ("C"), Lance ("L) and Jim ("J") are detailed in the following sections. Lance was in the same cohort as those completing the in-class tests. Unlike the others, Charlotte was a postgraduate student, but as she was in her first programming unit at the time of her think aloud session, she was at a similar level to those students in the in-class tests. Jim was in week 2 of his second programming unit.

In these excerpts, a pause in speech is marked "...", as a placeholder for dialog we have removed as it added nothing to the context of the think aloud session.

4 Test 1 (Week 2)

When the students completed Test 1 at the beginning of their week 2 lecture, they had completed two hours of lectures and a two hour workshop. The test questions are provided in the appendix. (We will hereafter refer to test questions in an abbreviated form. For example, Question 1 will now simply be Q1.) Our Test 1 is very similar to the Test 1 of Ahadi and Lister (2013), differing in only four respects: (a) our test is a translation from their Java to our Python, which is a trivial change given that all the questions in Test 1 are about assignment statements; (b) we renumbered their questions, (c) we omitted Q2a from the Ahadi and Lister test, but retained their Q2b as our Q7; and (d) we conducted our first test in week 2 whereas they conducted their first test in week 3.

Figure 1 shows the distribution of student scores on Test 1, where 8 is the maximum possible score.

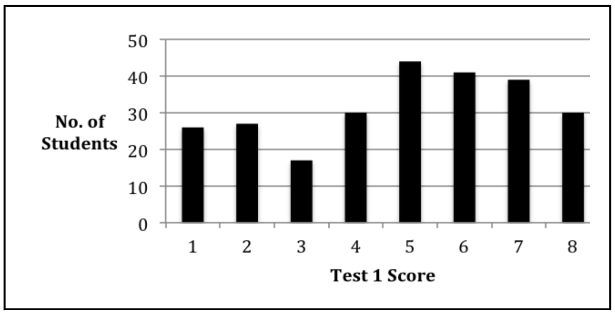


Figure 1: Distribution of total scores on Test 1 (N=254)

All questions were worth 1 point, with no fractional points awarded. Answers were treated as either right or wrong, but syntactic errors were ignored. We eliminated from Figure 1 and all subsequent analysis, the small number of students who scored zero on Test 1, as they were likely to be students who had not attended week 1 classes. As was the case for Ahadi and Lister (2013), there was a wide variation in Test 1 scores.

Table 1 shows the percentage of students, for each Test 1 score out of 8, who correctly answered each of the eight questions. The final row of the table represents the percentages of all students who answered correctly each question in the test. Cells containing asterisk/s indicate a statistically significant difference in the two percentages above and below the asterisk/s. (NB: percentages are rounded down.) As can be seen from that table (especially for test scores of 1 to 6 inclusive, as marked with darker border lines), an approximate rule of thumb is that if a student scored *n* points out of 8 on the test, then the student's first *n* answers were most commonly right, and their remaining answers were most commonly wrong. In accordance with that rule of thumb, we characterised the students as follows:

- Score 1 or 2: understands little of the semantics of the code.
- Score 3 or 4: applies inconsistent guessing because of fragile understanding of the semantics.
- Score 5: can conduct a trace with some reliability.
- Score 6: can perform inductive inference.
- Score 7: can sometimes perform deductive inference.

We elaborate on this characterisation in the next section.

4.1 Semantics of Assignment and Sequence

In Test 1, Q1–Q3 tested whether a student understood the semantics of a sequence of assignment statements. That is, the value on the right of the assignment is copied to the left, overwriting the previous value, and assignments are executed in sequence. Many students who scored 1, 2 or 3 on Test 1 struggled with Q1–Q3 (see the left three shaded columns in Table 1).

Test1 Score	*n*	semantics			tracing		reasoning		
		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	26	53	23	0	4	4	8	12	0
			**	**	*				
2	27	60	71	26	26	15	4	0	0
				***	*				
3	17	53	65	89	59	24	6	6	0
		*				*			
4	30	87	84	80	64	54	14	14	7
			*		**	**			
5	44	87	96	94	94	85	30	12	5
							***	*	***
6	41	86	98	98	96	88	69	35	35
								***	***
7	39	83	100	98	93	98	75	75	80
		*					**	**	**
8	30	100	100	100	100	100	100	100	100
all	254	78	84	76	73	65	43	34	32

Table 1: Percentage of students who answered correctly each part of Test 1, broken down by total score (χ^2, * is $p \leq 0.05$, ** is $p \leq 0.01$ and * is $p \leq 0.001$)**

Sensorimotor students often have no alternative but to use guessing as a strategy for reasoning about code. This is because they have not yet built a clear mental model of the notional machine (du Boulay 1989), nor do they have a solid comprehension of the concepts to which they have only just been introduced. Because of this, they inconsistently apply mis/conceptions about the semantics of code.

4.1.1 Vignettes from Charlotte

One of our think-aloud students, Charlotte, demonstrated this type of sensorimotor reasoning when she was asked to trace the effect of the three assignment statements (Q2) shown both in the appendix and again here in Figure 2.

```
r = 2
s = 4
r = s
```

Solution: `r` is 4, `s` is 4

Figure 2: Test 1 Q2 - Tracing Task

As Charlotte considered the code she said:

C: *Hmm. ... I don't know, but I imagine ... it's kind of a guess here [laugh], that ...* `r` *will equal 4 ... and* `s` *will equal 4.*

Of course students will get the marks for correct guesses in exams, and as this think aloud session showed, it is not until you listen to a student's reasoning that you can start to understand their true level of ability. This is consistent with the findings of Teague et al. (2012) who provided an astonishing contrast between the correct solution a programming student was able to produce and the inexplicable reasoning and method he actually used to produce that solution. This is of course the advantage of think alouds. It is quite obvious when a student flukes a correct answer. Think alouds also explain why, in other cases, students answer incorrectly.

With her very next task (Q3, shown again in Figure 3), Charlotte thought she was being consistent with her "guess", but that was not actually the case.

C: *So…going from how I did the last one, I might as well be consistent. ...* `p` *will equal 8 and* `q` *will equal 1.*

```
p = 1
q = 8

q = p
p = q
```

Solution: `p` is 1, `q` is 1

Figure 3: Test 1 Q3 – Tracing Task

Charlotte later reflected on that answer and explained:

C: *I looked up to the original integer rather than looking at the switched integer*

In other words, she looked only to the first assignment of `q` (i.e., `q = 8`) rather than taking account of its subsequent reassignment (`q = p`). Charlotte's fragile understanding of the semantics (as well as a floundering command of the jargon) is also exemplified in her next comment:

C: *I'm just not confident in how the rules of inheritance were applied. It was like I was just going on a whim.*

Students who scored 4 on Test 1 tended to answer Q1–Q3 correctly, and *either* Q4 or Q5 correctly. We characterise these students as novices who still have a fragile understanding of the semantics of the language, and like Charlotte, inconsistently apply mis/conceptions.

4.2 Inductive Reasoning

Lister (2011) proposed that a preoperational programming student can make reasonable inductive guesses about the function of a piece of code based upon the input/output behaviour they observe from tracing it, without understanding how the code achieves that function.

We have witnessed this type of reasoning in previous work (Teague et. al. 2013, Teague and Lister 2014a) where the student (Donald) attempted to explain the purpose of code that sorted the values in three variables. Donald based his answer on the effect of a single set of poorly chosen input values. As a result, his answer, although accurate for that single test case, did not reflect the purpose of the code for *any* set of input values.

The students described in this paper who scored 5 on Test 1 usually answered all the tracing questions correctly (Q1–Q5) but often could not explain the swap code they had just traced (Q6). In fact, Table 1 shows that out of the students who scored 5 on the test, only 12% of them could *explain* similar swap code (Q7); and only 5% of them could *write* similar swap code (Q8).

4.2.1 More Vignettes from Charlotte

Charlotte is illustrative of those students who can sometimes trace a piece of code but cannot explain that code. In her previous two tasks, Charlotte guessed, and applied inconsistently her misconceptions about assignment statements. It is not surprising, therefore, that her ability to reason about the *purpose* of code (Q6, shown in Figure 4) is very limited. This time, Charlotte traced the code accurately (or at least managed to guess the correct effect of assignment consistently), but she was unable to explain the code's overall purpose:

```
x = 7
y = 5
z = 0

z = x
x = y
y = z
```

Solution: The values in x and y were swapped

Figure 4: Q6 – Reasoning Task

C: *So if* `z` *equals* `x` *from above, that will become 7 ... If* `x` *becomes* `y` *...* `y` *is 5, so* `x` *becomes ... 5 ... If* `y` *equals* `z` *... it becomes 7, so I don't know what I observe.*

As shown in Table 1, of the students who scored 6 on Test 1, approximately two thirds of them made the correct observation for Q6, but only about a third could answer either Q7 or Q8.

Table 2 shows contingency tables for Q6 and Q7, and also Q6 and Q8, for those students who answered both Q4 and Q5 correctly. Most students who answered Q6 (explain swap by induction) incorrectly could not answer correctly either Q7 (explain swap by deduction) or Q8 (write swap). Even among students who did answer Q6 correctly, a substantial percentage could not answer correctly either Q7 or Q8. As a rough guide, answering Q6 correctly tends to be a necessary, but not sufficient, condition for answering Q7 and Q8 correctly.

Test 1 Q6 "what do you observe about final values in x and y" (induction)	Test 1 Q7 "explain swap" (deduction)		Test 1 Q8 "write swap"	
	wrong	right	wrong	right
wrong (n = 55)	26%	13%	30%	9%
right (n = 89)	28%	33%	25%	36%

Table 2: Contingency tables for Q6 & Q7 and Q6 & Q8, for students who answered both Q4 & Q5 correctly (χ^2, p= 0.012 for Q7 and p < 0.001 for Q8, N=144 for each of Q7 & Q8)

As noted above, Charlotte was one of those students who could not answer Q6 correctly. She was prompted by the interviewer to see that the code was swapping the values in variables `x` and `y`. She was then asked to explain the Q7 swap code, shown in Figure 5.

```
j = i
i = k
k = j
```

Solution: The values in i and k were swapped

Figure 5: Q7 – Reasoning Task

C: *when these lines of code are executed,* `j` *becomes ... is already* `i`*.* `i` *is* `k`*,* `k` *is* `j`*, so thereby ...* `j` *equals* `k` *which is already done at the end so I doubt that's right*

Perhaps Charlotte was reading the "=" as a statement of mathematical equality: if `j` is equal to `i`, and `i` is equal to `k`, then `j` is equal to `k`. However, the "=" operator is about assignment, not equality. In any event, Charlotte then shifted her reasoning about the code from being about statements of equality, to assigning values:

C: *Oh, well maybe ...* `j` *equals* `i`*,* `i` *equals* `k`*,* `k` *equals* `j` *...Yeah! well it takes away the need for* `i`*.*

Our interpretation of what Charlotte said is that `i` is not needed when swapping the values in `j` and `k`. In other words, a swap can be effected simply by assigning `k` to `j` and then `j` to `k`. Whatever her reasoning, we have seen that it is confused.

4.3 Deductive Reasoning and Code Writing

Lister (2011) proposed that deductive reasoning in programming was the ability to infer the computation performed by a piece of code, without needing to trace the code with specific values. Such ability is characteristic of the concrete operational stage in neo-Piagetian terms.

Students who scored 7 on Test 1 tended to answer all the tracing questions correctly (i.e. Q1–Q5) but tended to only answer correctly two questions out of Q6, Q7 and Q8, in near-equal percentages (75%, 75% and 80% respectively).

Table 3 shows the relationship between Q7 (explain swap by deduction) and Q8 (write swap) among the 144 students tested. Among these students, 24% of them could only answer one but not both of Q7 and Q8 correctly. However, a greater percentage of students who had explained the swap (Q7) could write a swap (Q8). This result is consistent with earlier findings by others that the ability to explain code is a prerequisite for the ability to write similar code (Lopez, Whalley, Robbins, and Lister 2008).

Test 1 Q7 "explain swap"	Test 1 Q8 "write swap"	
	wrong	right
wrong (n= 79)	43%	12%
right (n = 65)	12%	33%

Table 3: A contingency table comparing the performance of students on Q7 and Q8, for the students who answered both Q4 and Q5 correctly. (χ2, p < 0.001, N = 144)

4.3.1 Vignettes from Jim

Jim, another think-aloud student, had trouble with both Q7 and Q8, even after completing Q1–Q6 successfully. Jim looked at the code in Q7 (see Figure 5) and said:

J: `j` *has been changed ... to take the value of* `i` *... because* `j` *took the value of* `i`*, so* `k` *takes the value of* `j` *... therefore* `k` *is taking the value ... of* `i` *...*

Here, Jim used only the first and third lines of code in Figure 5 (and ignored the second line where `i` is reassigned) to reason about the value being assigned to `k`.

J: *so it's just a loop.*

By "loop" we believe Jim meant something about the movement of data between the variables rather than a looping control structure in the code. Jim's misconceptions about the assignments remained evident when he then took into account the second line of code, having considered the code in order of lines 1, 3 then 2:

J: So ... basically `k` *will keep its value and everything will become the value of* `k`*.*

In other words, his reasoning was: `j` is given the value of `i` (line 1); therefore `k` (in line 3) is taking the value of `i` too because it is assigned `j`; and `i`'s value originally came from `k`. So therefore, `k` is unchanged by this process, and the other variables both have the value of `k`. After the interviewer questioned Jim's summation (i.e. that `k` remained unchanged) he became less sure:

J: *No, the `k` will keep it's ... `j` will keep its value... no*

By this stage, Jim was confused and probably cognitively overloaded. He decided to restart the task and this time he wrote specific values for each of the variables. Resorting to tracing with specific values is typical behaviour for students who are yet to reach the concrete operational stage and who are weak at reasoning with abstractions.

J: *Ok, we'll just say ... we have `j` is equal to 1, `i` is equal to 2 and `k` is equal to 3.*

Jim traced the code again using those specific values which he wrote above the variables. However, he made a transposing error with the final line, causing him to assign `k`'s value to `j` instead of the other way around. His final trace of the three lines of code in Q7 (Figure 5) is shown in Figure 6.

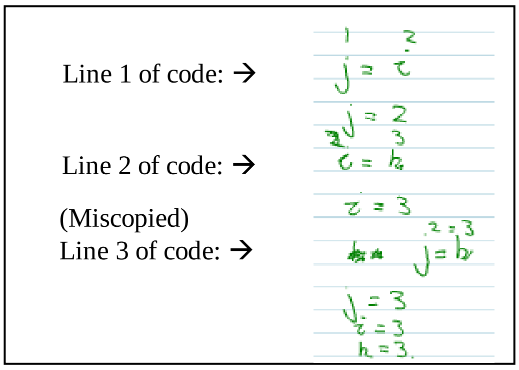

Line 1 of code: →
Line 2 of code: →
(Miscopied)
Line 3 of code: →
1 2
j = i
j = 2
3
i = k
i = 3
2 3
j = k
j = 3
i = 3
k = 3

Figure 6: Jim's trace of Q7

Jim was prompted to recheck this trace, and the interviewer suggested that a clearer way to articulate assignment was to say "is given" (rather than "is equal to") to help him focus on the direction of the assignment. Jim then corrected the miscopied assignment statement at line 3 in Figure 6 (to:"`k = j`"), but said:

J: *`k` is given to `j`, there we go*

Jim seemed to be getting confused between the direction of assignment (i.e. the movement between variables) and the articulation of the assignment statement (i.e. reading left to right). So the interviewer ("I") intervened further:

I: *No. `k` is assigned the value of `j`. So `j` is given to `k`. Depends which way you want to read it. ...*

J: *Yeah, so ... `j` becomes `k`.*

I: *No. in this case, k becomes j*

J: *oh, `k` becomes `j` sorry ... so `k` is equal to 2.*

Given the difficulties with assignment that Jim manifested here in Q7, it is surprising that Jim managed to answer Q1 to Q6 correctly. We speculate that Jim's problems here are due to the higher cognitive load.

Finally having traced the code correctly, Jim attempted to explain its purpose. This proved even more difficult:

J: *it's just really reassigning. Isn't it? Because we have `j` is equal to 2, `i` is equal to ... 3 and `k` is equal to 2.*

Jim's response is a vague overview of the code, equivalent to "all the variables have been changed". Asked if the code was doing something similar to that in the example in Q7 he replied:

J: *it's similar, in the sense that it's swapping ... um, we've got .... `c` becomes `a` ... `a` becomes ... `b` and `b` becomes `c`, so that's just swapping them*

In terms of the SOLO taxonomy (Biggs and Collis 1982) this is a multistructural answer – recounting the effect of each individual line, rather than the total effect of all three lines. Asked which variables are swapped:

J: *the first ones ... `j` swapped, `j` took the value of `i` ... `i` and `j` swapped*

It is clear now that what Jim meant by "swap" was "change", rather than a two-way exchange of values. After clarification of what a "swap" was, and looking at what each of the variables started and ended up with, Jim was finally able to answer that indeed there had been a swap of values between two variables:

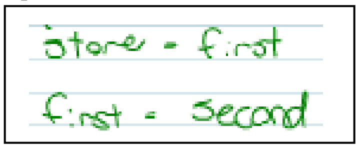
J: *apparently `i` swapped with `k`*

Jim's use of the word "apparently" suggests a lack of conviction. His difficulty with the tracing task showed misconceptions which are characteristic of novices at the sensorimotor stage. However, sensorimotor novices are also reluctant to retrace as it is a cognitively demanding task given their fragile domain knowledge. But Jim decided to redo the task, this time in a manner he was more comfortable with. He introduced specific values. Novices at the preoperational stage are unable to deal solely with abstractions and require specific values to make sense of code. In terms of the Overlapping Waves Model (as described Section 2), we suggest that Jim is in the process of developing preoperational skills, while still displaying some legacies of the sensorimotor stage.

4.3.2 Vignettes from Lance

After seeing how Jim dealt with reasoning about three lines of assignment statements, the reader will not be surprised that he had difficulty writing similar code. In fact (as shown in Table 1) 20% of the students who scored 7 correctly answered all of preceding tracing and reasoning questions (Q1–Q7) but then could not *write* similar code (Q8).

Our final think aloud student, Lance, had difficulty writing the code, even though he had answered Q1–Q7 correctly. For Q8, Lance wrote the first (correct) lines of code to swap the variables `first` and `second`:



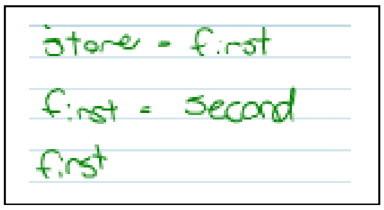
```
store = first
first = second
```

Figure 7: Lance's 1st two Lines of Q8 Swap Code

But his explanation of that code was inaccurate:

L: *ok so now ...* `second` *should have the number that* `first` *has in it*

Lance had written the assignment statement in one direction and articulated it in the opposite direction. He continued with the third line of code before hesitating:

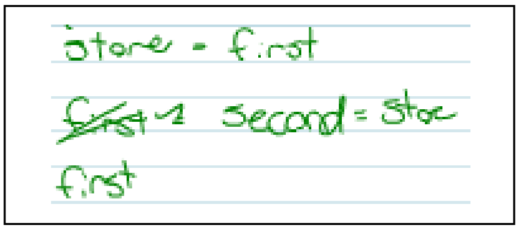


store = first
first = second
first

Figure 8: Lance's 3rd Line of Q8 Swap Code

L: *oh no that's wrong ... I think ... that <u>is</u> wrong because ... um ... ok it should be* `second` *equals* `store` *... shouldn't it*

Lance changed his code to:



store = first
~~first =~~ second = store
first

Figure 9: Lance's Revised 2nd Line of Q8 Swap Code

After reading his revised code, Lance decided to start again. Like Jim and other novices reasoning at the preoperational stage, this time he used specific values to help him reason about the code he was writing.

L: *ok so you've got ... let's just say that's 1 and that's 2 so I can keep it in my head. ok this will make it a bit easier alright*

While Lance assigned the values 1 and 2 to variables `first` and `second`, writing the code still proved not to be straight forward:

L: *so first you're going to need to store the ... memory of* `first` *... like the number in* `first` *... so we're gunna go ...* `store` *... equals* `first` *...*

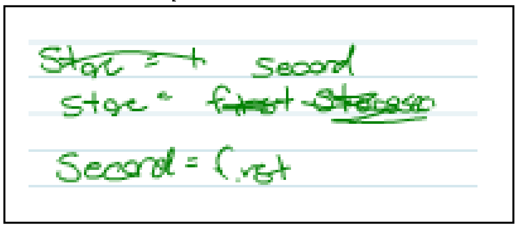
Although Lance said "`store` ... equals `first`" he wrote "`store = 1`". We don't believe he meant to write "1", but he was no doubt thinking that `first` had the value 1. He was working at the preoperational level at which it is difficult to reason in abstract terms. In any event, he quickly self-corrected this error by changing the code to "`store = first`".

Lance then gave an confused explanation of what the code needed to do:

L: *ok ... just stored ... the number from* `first` *into ...* `store` *... then you go from ... we need to put the number that <u>was</u> in* `first` *into* `second` *so if we go ... because we're stored* `first` *we can put ... that in there because it's remembered now ... so if we go ...* `first` *equals* `second` *... I think ... no that's what I was doing before ... and I thought it was wrong ... maybe if we just store* `second`

Lance sought confirmation from the interviewer that it would in fact make no difference whether he stored the value of `first` or `second` to begin with. He decided to make the change anyway, although he wrote by mistake "`store = stores`". After fixing this error he said:

L: *ok so* `store` *equals* `second` *... why is it so confusing it's so simple [laugh] confusing ... alright* `store` *equals* `second` *so you go* `store second` *and then ... that number's remembered ... and that's 2 ... and basically we want to assign that ... to ... we want to assign* `first` *... alright we want to overwrite the 2 in* `second` *... to the 1 in* `first` *so if we go ... um ...* `second` *equals* `first`



~~Store = first~~ second
store = first ~~stores~~
Second = first

Figure 10: Lance's 2nd Attempt at Q8 Swap Code

Although he made no note of the changing values on paper, Lance constantly used specific values to talk about the effect of the assignments. He seemed unable to cope with even the abstraction of variable names. As he said before, using specific values makes it easier for him "to keep in his head". And this tactic eventually worked.

L: *so now you've got ... ah the 1 in* `second` *... and the 2 in* `store` *and then if you go* `first` *equals* `store`*...*

In summary, when it came to writing code in Q8, Lance struggled to implement code very similar to code he had just successfully traced and reasoned about. He failed to write code until he introduced specific values, which enabled him to visualise the changing values in the variables. Preoperational novices are reliant on specific values to reason about and write code.

Only 30 students (12%) who completed Test 1 scored the maximum possible 8 marks, and were deemed competent at tracing, reasoning about and writing very simple code. Given their consistent correct performance, these students are unlikely to have been guessing about the semantics of the code. The fact that they were also able to write the code in Q8 would lend us to believe that they were at least operating at the preoperational level. While these students may be reasoning at the concrete operational stage we are reluctant to draw that conclusion with confidence, without knowing how they went about solving the problems, given the evidence of superficially correct solutions presented by Teague et al (2012).

5 Test 2 (Week 4)

We conducted our second test two weeks later, in week 4.

5.1 Test 2 Q1 (tracing question)

This first question in Test 2 was a tracing question equivalent to the last tracing question in Test 1 (Q4). Students who scored 1–4 in Test 1 tended to perform poorly on the last tracing question in that same test (Q4, see Table 1). However, all students performed very well on the first tracing question in Test 2, with the probability of answering this question at 77% for those who scored 2 in Test 1, and at 96% for all other students. So the students who had lagged behind on tracing skills in week 2 had substantially closed the gap by week 4, at least on this type of question.

5.2 Test 2 Q2 (writing question)

The second question in Test 2 was exactly the same as Q8 in Test 1. That is, the students were required to write code to swap the values in two variables, `first` and `second` (see appendix).

Figure 11 plots the probability of students answering this Test 2 question correctly, against their total score on Test 1. The largest circle in Figure 11 represents 26 students, while the smallest circle represents 10 students.

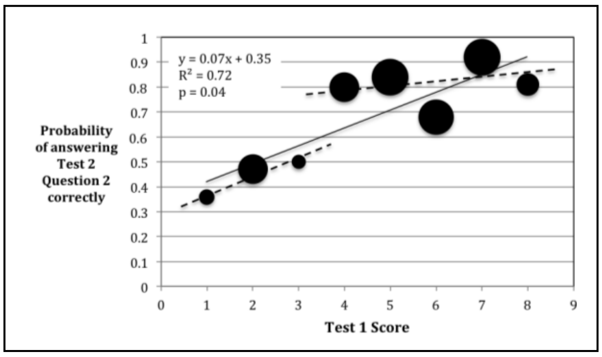


Figure 11: Relationship between Test 1 scores and the probability of answering Test 2Q2 correctly (N=156)

The solid regression line shown in Figure 11 accounts for 72% of the variation, and that regression line is statistically significant at the 0.05 level. Therefore overall performance on Test 1 (week 2) is a good predictor of performance on this code writing question in the week 4 test (Test 2, Q2). Recall from the previous subsection, however, that performance on the week 2 test was not a good predictor of performance on the week 4 tracing question (Q1), so we cannot conclude simply that students who do better on Test 1 tend to do better on all questions in subsequent tests.

Inspection of Figure 11 suggests that, although the solid line of regression is a good predictor, there does appear to be a non-linear jump in performance between students who scored 1–3 on Test 1 and students who scored 4–8. The two dashed lines are lines of regression through each of those two groups of students, and serve to highlight that possible performance gap. Note, however, that neither of these two dashed regression lines meets the traditional 0.05 statistical criterion for significance, perhaps because of the small sample size. This possible performance gap suggests that, while students who scored 1–3 on Test 1 have closed the gap on tracing skills for these simple tracing problems, they have not closed the gap on deductive and code writing skills. That is, while students who scored 1–3 on Test 1 are progressing in their learning, they are not progressing as quickly as students who scored higher on Test 1. Our interpretation of this in neo-Piagetian terms is that the students who scored 1–3 on Test 1 were now better at tracing code, but they were still operating (at most) at a preoperational level of reasoning. They had not made the transition to the concrete operational stage. They remained unable to reason about abstractions and therefore unable to write simple code.

6 Test 3 (Week 7)

Our third test was conducted in week 7, five weeks after the first test. By this stage of semester, students had been introduced, amongst other concepts, to conditional statements and Python lists.

6.1 Test 3 Q1 (swapping list elements)

Figure 12 shows the first question from Test 3, which also requires students to write a swap, but in this case it is a swap between two elements of a Python list.

A list called `ages` has been created in Python. There are two values out of order in the list and these values are stored at indexes 0 and 2. Write code to swap those two values so that the list would be in order.

Sample Solution:

```
temp = ages[0]
ages[0] = ages[2]
ages[2] = temp
```

Figure 12: Test 3 Q1 with sample solution

Figure 13 plots the probability of students answering Test 3 Q1 correctly, against their total score on Test 1. The largest circle in Figure 13 represents 18 students, while the smallest circle represents 4 students.

While the regression in Figure 13 does show a statistically significant linear relationship ($p < 0.01$), there is a clear non-linearity in the neighbourhood of the Test 1 score of 5. A non-parametric χ^2 test shows that the gap between scores of 5 and 6 is statistically significant at the 0.1 level (see Table 4).

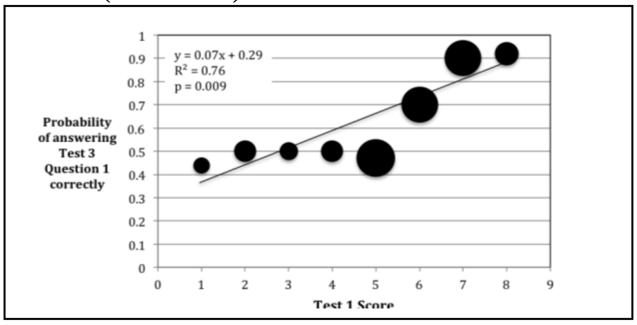


Figure 13: Relationship between Test 1 scores and the probability of answering Test 3 Q1 correctly (N=117)

Thus students who could not perform inductive inference (i.e. those operating at the sensorimotor level) in the week 2 test are, 5 weeks later, still tending to reason at the sensorimotor level, and lag behind those students who could perform inductive inference (i.e., those operating at least at the preoperational level) in week 2.

Test 1 score	N	Test 3 Q1	
		Wrong	Right
5 (i.e. typically could trace with some reliability in Test 1)	21	52%	48%
6 (i.e. typically could perform inductive inference in Test 1)	20	30%	70%

Table 4: A contingency table comparing students on Test 1 scores 5 & 6 versus Test 3 Q1 (χ^2, p=0.1, N=41)

The gap between Test 1 scores of 6 and 7 is also statistically significant at the 0.1 level (see Table 5).

Students who could not perform deductive inference (at best, preoperational) in the week 2 test are, 5 weeks later, still lagging behind those students who could perform deductive inference (concrete operational) in week 2.

Test 1 score	N	Test 3 Q1	
		Wrong	Right
6 (i.e. typically could perform inductive inference in Test 1)	20	30%	70%
7 (i.e. could sometimes perform deductive inference in Test 1)	20	10%	90%

Table 5: A contingency table comparing students on Test 1 scores 6 & 7 versus Test 3 Q1 (χ^2, p=0.1, N=40)

7 Test 4 (Week 9)

We conducted a final test in week 9. One of the questions required students to write code to swap values in a list. On this occasion the values in the list were to be swapped only if they were out of order. The only students who did well with this question were those who scored 100% on Test 1. For all other students, the probability of getting it right was less than 50%.

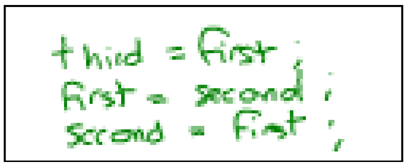
Among those who scored 1 to 7, there appears to be a performance gap on this question with students who performed very poorly on Test 1 (29% probability for Test 1 scores 1–3) performing considerably worse than the students who demonstrated some ability to trace reliably in Test 1 (49% for scores 4–7).

8 Charlotte's Progress

We have so far seen that Charlotte struggled in Test 1 to both trace and explain simple assignment statements. In neo-Piagetian terms this means she was likely reasoning at the sensorimotor stage. Not surprisingly, she also failed the concrete operational task of code writing in that same test. She hypothesised that a third variable would be required in order to make a swap, referring to the code shown in the previous question (Test 1 Q7, see appendix).

C: I'll follow the format from above ... 'cause it makes sense 'cause it worked

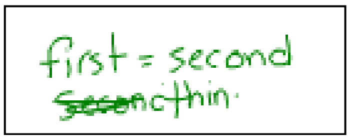
Her strategy was to give each of the variables a value, and she noted what their values should be once her code had executed. Then she wrote the incorrect code in Figure 14.



```
third = first ;
first = second ;
second = first ;
```

Figure 14: Charlotte's First Attempt in Week 2

When Charlotte attempted the very same code-writing task five weeks after her first think aloud, she still struggled with it. She initially failed to use a third (temporary) variable, as can be seen from the first line of code in Figure 15. For the second line, she started writing "second", crossed it out and replaced it with (an incomplete) "third" before crossing out all that she had written (shown in Figure 15).



```
first = second
```
~~Seconcthin~~·

Figure 15: Charlotte's Second Attempt in Week 7

Charlotte almost immediately then wrote correct code, and verified her solution using specific values for `first` and `second`. Charlotte was now, five weeks after the first think aloud, working at the preoperational level: having overcome her initial misconceptions, she was able to trace and write very simple, familiar, code.

Two weeks later, Charlotte completed Test 4 before we had a think aloud session with her. Her final code for a conditional swap of list elements was accurate. However, when she reflected on this question in a subsequent think aloud session, Charlotte confessed to not being sure of the correctness of her solution and voiced some confusion about assigning array elements:

C: I was thinking `temp` had to be an array…

Having previously developed the ability write swap code, Charlotte was then manifesting misconceptions with less familiar material: arrays. Her behaviour is consistent with an Overlapping Waves Model, where the introduction of a new concept may result in reversion to a less mature stage (for that concept).

9 Conclusion

Our think aloud excerpts have answered the first of the questions posed earlier, regarding the strategies, behaviour and misconceptions that are evident in novice programmers. We categories these (in Table 6) using the neo-Piagetian (NP) framework (where SM=sensorimotor; Preop=preoperational).

Behaviour	**NP Stage**
guessing	SM
fragile grasp of semantics	SM
confused use of nomenclature	SM
inability to trace simple code	SM
misconceptions (about sequence, assignment, mental models and the notional machine)	SM
errors due to cognitive overload	SM/Preop
reluctance to trace	SM/Preop
ability to trace but not explain code	Preop
reliance on specific values	Preop

Table 6: Novice Programmer Behaviour

Next, we address each of the remaining questions:

Why do students get particular questions wrong?

There are a number of reasons, including guessing, misconceptions, inability to work with abstractions; and inability to focus on more than one element of a scenario.

Can a student have disparate levels of ability with two tasks which test similar programming concepts?

This behaviour was in fact evident with the tasks requiring students to trace code, then to reason about its purpose. A preoperational student can trace code, but they do not develop the ability to reason about its purpose until the concrete operational stage.

Why are some students unable to work with abstractions?

Ability to work with abstractions is not evident until the concrete operational stage. Based on our quantitative results , only the 12% of students who answered all the week 2 test questions correctly were likely to be reasoning at the concrete operational stage at that time, and only those students were manifesting concrete operational skills late in semester.

These results are consistent with our previous studies (Ahadi and Lister 2013; Ahadi et al. 2014) and means that most students are still manifesting sensorimotor and preoperational reasoning at the end of their first semester. Our think aloud studies support this. These results suggest that introductory programming educators are underestimating the foreignness to students of concepts taught very early in semester as well as their inability to reason abstractly.

10 Pedagogical Discussion

While it may be up to each student to practise and improve within a neo-Piagetian stage, we believe the teacher's role is to assist the students to transition from one neo-Piagetian stage to the next. We now offer suggestions on how they might facilitate that. As a general rule we agree with Bruner (1960):

It is into the language of (the novice's) internal structures that one must translate ideas if the (novice) is to grasp them.

10.1 From Sensorimotor to Preoperational

A sensorimotor student who guesses cannot be aware of *which* reasoning is accurate without external feedback. Until they have external feedback they are unlikely to resolve their misconceptions. Teachers should facilitate environments that encourage deliberate, supported practice (Guzdial 2014). We speculate that students who have not had external feedback "hedge their bets" in exams in the hope that one of the strategies is correct and will at least get them part marks.

Teachers should begin by offering students one-liner single-concept tasks. The earliest tasks should be purely literal expressions with gradual progression to univariate expressions. Teachers should be aware of and discourage rote learning and pattern matching, as that delays the transition to a higher stage.

Teach students how to trace code systematically, for example with a trace table, using appropriate values (test categories and cases). Furthermore, test them to ensure that they *are* tracing correctly.

Students at the sensorimotor stage require, more than anything else, that their misconceptions are corrected. For example: "what is an assignment statement?" or "what can (and can't) a variable do?". When students have overcome any misconceptions (especially about variables, assignment and sequence) and have a clear idea of the notional machine, and can start to trace code reliably, they are probably reasoning at the preoperational stage.

10.2 From Preoperational to Concrete

Teachers should gradually increase the complexity of the tasks with multivariate expressions and more complex code. Roles of variables (Kuittinen and Sajaniemi 2004) is one example of useful cognitive concepts that encourage abstract reasoning. In general, there should be a focus on tracing and explaining tasks with code writing tasks secondary.

10.2.1 Tracing and Explaining Code

Give preoperational students a complete function or very small program that does something interesting – perhaps with visual impact. Set them the task of experimenting with the code by making small, superficial changes. Give them practice at interpreting the results of a trace (i.e., identifying invariants and explaining the code's overall purpose). A good assessment task at this stage is to supply "buggy" code where the skills students have developed (above) are used to fix the code.

10.2.2 Abstract Tracing

Preoperational students are heavily reliant on specific values in variables to reason about code. This reliance diminishes as they become more proficient with programming and they develop an ability to trace "abstractly". In other words they are able to compute the effect of the code without using specific values. This ability to start working with abstractions signals the transition into concrete operational reasoning. Jim, for example, tried unsuccessfully to trace code abstractly (i.e., without specific values). However, he then succeeded by resorting to the use of specific values. He, and other preoperational students, will develop abstract tracing skills with persistent practice and challenges that require more mature strategies until they learn to reason about and work with abstractions. Tracing abstractly also means that the trace need not be complete in order to determine the code's purpose. A student transitioning into concrete operational stage may be able to short-circuit a trace because they can also simultaneously process a number of features of a block of code (e.g., in a loop). Only once students have begun to develop those sorts of reading skills will they begin to write code systematically.

11 References

Ahadi, A. and Lister, R. (2013): Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant? Proc. of *Ninth Annual International ACM Conference on International Computing Education Research (ICER 2013)*, San Diego, CA. 123-128, ACM.

Ahadi, A., Lister, R. and Teague, D. (2014): Falling Behind Early and Staying Behind When Learning to Program. Proc. of *Psychology of Programming Interest Group (PPIG 2014)*, Brighton, UK.

Biggs, J. B. and Collis, K. F. (1982): Origin and Description of the SOLO Taxonomy *Evaluating the quality of learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press Inc.

Boom, J. (2004): Commentary on: Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 239-247.

Bruner, J. S. (1960): *The Process of Education*. London: Oxford University Press.

du Boulay, B. (1989): Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* 283-300. Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A. and Simon, H. A. (1993): *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.

Falkner, K., Vivian, R. and Falkner, N. J. G. (2013): Neo-Piagetian Forms of Reasoning in Software Development Process Construction. Proc. of *Learning and Teaching in Computing and Engineering (LaTiCE) 2013*, Macau. IEEE.

Feldman, D. H. (2004): Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 175-231.

Guzdial, M. (2014). Anyone Can Learn Programming: Teaching > Genetics. BLOG@CACM http://m.cacm .acm.org/blogs/blog-cacm/179347-anyone-can-learn-programming-teaching-genetics/fulltext 2014.

Kuittinen, M. and Sajaniemi, J. (2004). Teaching Roles of Variables in Elementary Programming Courses. ITiCSE '04. Leeds, UK, ACM.

Lister, R. (2011): Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. Proc. of *13th Australasian Computer Education Conference (ACE 2011)*, Perth, WA. **114:**9-18, ACS.

Lopez, M., Whalley, J., Robbins, P. and Lister, R. (2008): Relationships between Reading, Tracing and Writing Skills in Introductory Programming. Proc. of *ICER '08*, Sydney, Australia. ACM.

Siegler, R. S. (1996): *Emerging Minds*. Oxford: Oxford University Press.

Teague, D., Corney, M., Ahadi, A. and Lister, R. (2013): A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers. Proc. of *15th Australasian Computing Education Conference (ACE 2013)*, Adelaide, Australia. **136:**87-95, ACS.

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A. and 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. Proc. of *Australasian Association for Engineering Education Conference (AAEE 2012)*, Melbourne.

Teague, D. and Lister, R. (2014a). Longitudinal Think Aloud Study of a Novice Programmer. *Australasian Computing Education Conference (ACE 2014).* Auckland, New Zealand, ACS. **148**.

Teague, D. and Lister, R. (2014b): Blinded by their Plight: Tracing and the Preoperational Programmer. Proc. of *Psychology of Programming Interest Group (PPIG) 2014*, Sussex, UK.

Teague, D. and Lister, R. (2014c): Manifestations of Preoperational Reasoning on Similar Programming Tasks. *Australasian Computing Education Conference (ACE 2014).* Auckland, New Zealand. **148**, ACS.

Appendix: The Test 1 Questions

Q1 In the boxes, write the values in the variables after the following code has been executed:

```
a = 1
b = 2
a = 3
```

The value in `a` is $\boxed{3}$ and the value in `b` is $\boxed{2}$

Q2 In the boxes, write the values in the variables after the following code has been executed:

```
r = 2
s = 4
r = s
```

The value in `r` is $\boxed{4}$ and the value in `s` is $\boxed{4}$

Q3 In the boxes, write the values in the variables after the following code has been executed:

```
p = 1
q = 8

q = p
p = q
```

The value in `p` is $\boxed{1}$ and the value in `q` is $\boxed{1}$

Q4 In the boxes, write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 3

x = y
z = x
y = z
```

The value in `x` is $\boxed{5}$ `y` is $\boxed{5}$ and `z` is $\boxed{5}$

Q5 In the boxes, write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 0

z = x
x = y
y = z
```

The value in `x` is $\boxed{5}$ `y` is $\boxed{7}$ and `z` is $\boxed{7}$

Q6 In Q5 above, what do you observe about the final values in x and y? Write your observation (in one sentence) in the box below.

Sample solution: The values in `x` and `y` were swapped.

Q7 The purpose of the following three lines of code is to swap the values in variables `a` and `b`, for any set of possible values stored in those variables.

```
c = a
a = b
b = c
```

In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables. Assume that variables i, j and k have been declared and initialised.

```
j = i
i = k
k = j
```

Sample solution: Swaps the values in `i` and `k`.

Q8 Assume the variables `first` and `second` have been initialised. Write code to swap the values stored in `first` and `second`.

Sample solution:

```
temp   = first
first  = second
second = temp
```