Longitudinal Think Aloud Study of a Novice Programmer

Donna Teague
Queensland University of Technology,
Brisbane, QLD, Australia
Tel: +61 7 3138 2000

d.teague@qut.edu.au

Raymond Lister
University of Technology, Sydney,
Sydney, NSW, Australia
Tel: +61 2 9514 1850

Raymond.Lister@uts.edu.au

Abstract

Recent research from within a neo-Piagetian perspective proposes that novice programmers pass through the sensorimotor and preoperational stages before being able to reason at the concrete operational stage. However, academics traditionally teach and assess introductory programming as if students commence at the concrete operational stage. In this paper, we present results from a series of think aloud sessions with a single student, known by the pseudonym "Donald". We conducted the sessions mainly over one semester, with an additional session three semesters later. Donald first manifested predominately sensorimotor reasoning, followed by preoperational reasoning, and finally concrete operational reasoning. This longitudinal think aloud study of Donald is the first direct observational evidence of a novice programmer progressing through the neo-Piagetian stages.

Keywords: Neo-Piagetian theory, programming, think aloud.

1 Introduction

Using neo-Piagetian theory, Lister (2011) conjectured there were four main stages of cognitive development in the novice programmer, which are (from least mature to most mature):

Sensorimotor: The novice programmer cannot reliably manually execute a piece of code and determine the final values in the variables (i.e., "trace" code). This incompetence is due both to misconceptions about programming language semantics and the inability to organise a written trace. Without the ability to trace accurately, and thus having no real capacity to check their own code, these novices can write incoherent code.

Preoperational: The novice can trace code reliably, but struggles to "see the forest for the trees". That is, the novice struggles to understand how several lines of code work together to perform some computational process. When trying to understand a piece of code, such novices tend to use an inductive approach. That is, they may perform one or more traces with differing initial values, and make an educated guess based on the input/output behaviour. These novices also struggle to see the relationship between diagrams and code. When writing code, these novices tend to patch and repatch their code, on the basis of their results from tracing specific values through their code. They cannot truly design a solution.

Concrete operational: The novice programmer is capable of deductive reasoning. That is, the novice can understand short pieces of code by simply reading the code, rather than tracing with specific values. When reading code, they can abstract from the code itself to reason in terms of a set of possible values that each variable may have. These novices can design code, at least for algorithms that can be easily visualized as diagrams. However, novices at this stage tend to only reason about relatively short pieces of code that perform relatively familiar computational processes.

Formal Operational: Writing programs is frequently referred to as an exercise in problem solving. McCracken *et al*. (2001) defined problem solving as a five step process: (1) abstract the problem from its description, (2) generate sub-problems, (3) transform sub-problems into sub-solutions, (4) recompose, and (5) evaluate and iterate. It is only at the formal operational stage that novices can reliably and efficiently perform problem solving.

Levin (1986, p. viii) summarised the general change in the novice through these four stages (in any domain, not just programming) as being a process of:

1. Increasing logical-mathematical power;
2. Differing modes of representations – from perceptual to formal;
3. Increasing attentional scope and integrational ability; and
4. Increasing skill with applying the competencies of lower stages, along with the adoption of new strategies.

Corney *et al*. (2012) provided indirect evidence that novice programmers pass through the preoperational and concrete operational stages, by analysing student answers to questions in an end-of-semester exam. They found that (a) within individual exam questions, there were students who could provide a preoperational answer but not a concrete operational answer, and (b) across exam questions, students tended to consistently provide either a preoperational answer or a concrete operational answer. However, such indirect evidence does not indicate the actual thought processes of a student.

In this paper, we provide direct evidence that a student passes though these neo-Piagetian stages. We had several volunteer students complete programming related tasks while "thinking aloud" (Ericsson and Simon 1993). We met approximately once each week with these volunteers, so we could follow their progress over the course of a

Classical Piagetian Theory	Vs.	Neo-Piagetian Theory
Is concerned with the general cognitive development of children.	vs.	Is concerned with the cognitive development of people of any age as they learn any new cognitive task.
A child at a particular Piagetian stage applies the same type of reasoning to all cognitive tasks (e.g., math and chess), apart from exceptions known as *décalage*.	vs.	Since a person's cognitive ability in any domain is a function of their domain knowledge, a person will often exhibit different Piagetian stages in different knowledge domains. Hence … *<continues in next row of this column>*
General tests, such as the pendulum test (Inhelder and Piaget 1958; Bond, 2005), can determine the Piagetian stage of an individual.	vs.	… there are no general tests, thus the failure to find strong correlations between programming ability and the pendulum test (e.g. Bennedsen and Caspersen 2008).
Prescribes typical age ranges for each Piagetian stage, but empirical evidence shows great flexibility in age ranges, due to cultural and environmental factors (Cole 1996, pp. 86-92).	vs.	The time that individuals spend in any stage is free to vary, and varies according to their rate of knowledge acquisition in a specific knowledge domain.
Children spend an extended period in one stage, before undergoing a rapid change to the next stage – the "stair case metaphor".	vs.	The staircase metaphor is sometimes applied, but also so is the "overlapping wave" metaphor (Siegler 1996) – see Figure 1 and section 1.2.

Table 1: Classical versus Neo-Piagetian Theory

semester. This paper documents the progress made by one student, who we refer to as "Donald" (a pseudonym). Donald is a male student, who speaks English as his first language. At the time our study began, Donald was 22 years old, and he was in his second semester of learning to program. However, Donald's first semester course was a breadth first introduction to computer languages (including SQL and HTML), and only 75% of the course was concerned with programming. We found Donald's behaviours so interesting, and his interest in our study so high, that we continued to conduct think alouds with him beyond that initial semester.

Before describing the think alouds with Donald in the next section, the remainder of this introduction will discuss three aspects of the framing of our research: (1) the nature of neo-Piagetian theory versus the better known classical Piagetian theory, (2) the justification of the neo-Piagetian framework over both Bloom and SOLO, and (3) the nature and purpose of our qualitative research.

1.1 Classical versus Neo-Piagetian Theory

It is well known that researchers since Piaget have conducted experiments that call into question aspects of "classical" Piagetian theory. Less well known, however, is that modifications to Piaget's classical theory have been proposed that address those experimental findings. One set of modifications is known as neo-Piagetian theory. (The "neo" is increasingly inaccurate, given that this "new" Piagetian theory is already several decades old.) Table 1 summarises some of the differences between classical and neo-Piagetian theory. For longer treatments of classical and neo-Piagetian theory, the reader is referred elsewhere (Demetriou, Shayer and Efklides 1992; Feldman 2004; Flavell, Miller, and Miller 2001; Lourenco and Machado 1996; and Sutherland 1992). In the next subsection, we will elaborate on the final row of Table 1, given that the concept of stages as overlapping waves is central to the empirical findings of this paper.

1.2 Stages as Overlapping Waves

Perhaps no aspect of classical Piagetian theory has generated more debate than the concept of stages. In classical Piagetian theory, children spend an extended period in one stage, before undergoing a rapid change to the next stage. Having made that change, children do not regress to the earlier stage. This is commonly referred to as the "*stair case metaphor*". The stair case metaphor suffers from two broad types of problems. The first problem type is empirical – people have been observed to exhibit simultaneously the reasoning patterns of more than one stage. The second problem type is philosophical – how and why does a person make the quantum leap from one stage to the next? While some neo-Piagetian researchers still accept the stair case model, others have found evidence for the "*overlapping wave*" metaphor (Siegler, 1996; Feldman, 2004; Boom, 2004). That metaphor is illustrated in Figure 1.

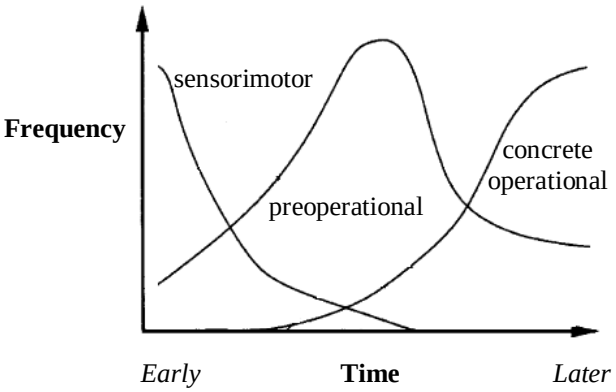

Figure 1: The Overlapping Wave Model.

According to the overlapping wave metaphor, as a person acquires knowledge in a new cognitive domain, the person exhibits a changing mix of reasoning strategies from different stages. Initially, the sensorimotor stage of reasoning is dominant, but its frequency of use declines. As the sensorimotor stage declines, there is an increase in the use of preoperational styles of reasoning, which

becomes dominant, before it in turn gives way to concrete operational reasoning. Not shown in Figure 1 is formal operational reasoning, which would develop in the same way. As will be apparent when we present the think aloud data for Donald, his progression fits the overlapping wave model.

1.3 Piaget vs. Bloom and SOLO

There have been earlier taxonomic descriptions of programming tasks, especially programming exam questions, based upon the popular Bloom's Taxonomy (Bloom et al. 1956; Whalley et al. 2006) and the SOLO taxonomy (Biggs and Collis 1982; Lister et al. 2010). In this section we briefly justify our use of neo-Piagetian theory in preference to Bloom and SOLO.

Bloom is a taxonomy of questions, not a taxonomy of possible answers. That is, a question must be classified as belonging to a single level of Bloom's taxonomy, and if a question is classified as being in one of the lower four levels of Bloom's taxonomy, there is only a single binary decision to be made about a novice's answer to that question − whether the answer is satisfactory or unsatisfactory at the prescribed Bloom level. Bloom is not suited to analysing questions where a population of novices may provide a rich variety of qualitatively different answers to a question. Nor does Bloom provide any mechanism for analysing think aloud data generated from the process by which a novice arrived at an answer.

The SOLO taxonomy is intended for classifying a rich variety of qualitatively different written responses to a question. However, SOLO does not provide any mechanism for analysing think aloud data generated from the process by which a novice arrived at a response. Biggs and Collis made a conscious design decision that SOLO was only for application to analysing final responses, not the mental process leading to that response (see pp. 21−23). Their reason for that decision was that they derived SOLO from classical Piagetian theory, and the restriction of SOLO to analysing final responses was their approach to avoiding the problems with classical Piagetian theory that were discussed earlier in this paper. Note that Biggs and Collis published SOLO in 1982, before almost all the developments in neo-Piagetian theory that provide an alternative way of avoiding the problems with classical Piagetian theory. Unlike SOLO, neo-Piagetian theory preserves the Piagetian mechanisms for analysing think aloud data generated from the process by which a novice arrived at a response.

Furthermore, given SOLO's focus on responses to questions, and the conscious exclusion from SOLO of the process by which a response is generated, SOLO does not lend itself to generating ideas for questions to put to students in think aloud sessions, whereas neo-Piagetian theory (through concepts such as reversibility, conservation and transitive inference) has proven to be a rich source of inspiration for us. All the problems we put to Donald (apart from tracing problems) were inspired by problems that Piaget used on children. The observations that Piaget made using his problems also provided strong suggestions as to what to look for in Donald's think aloud sessions.

We regard our use of neo-Piagetian theory as a logical progression from the earlier research that used SOLO.

1.4 N = 1?

Some readers may be disturbed by our small sample size – a single student. To argue for a larger sample, however, is to argue from a positivistic perspective, which is not a wrong perspective, but it is a perspective orthogonal to the aims of this paper. Our research is qualitative, not quantitative. That is, our aim is to identify some aspects of the nature of how novices reason about programs. Our aim is to neither identify all aspects of how novices reason about programs, nor to count the frequency with which a particular aspect occurs in a population of novice programmers.

Our use of think aloud sessions is an example of the microgenetic research method, which has been applied in many domains to test theories of cognitive development, and which is defined as having three main properties (Siegler 2006, p 469):

1. Observations span the period of rapidly changing competence.
2. The density of observations is high, relative to the rate of change. In the first semester of this study, think aloud sessions were conducted once a week (although for space reasons we only report three such sessions in this paper).
3. Observations are analysed intensively, to infer the representations and processes used by the students.

The microgenetic method has been used previously by Lewis (2012) to study a single novice programmer. We regard our research, and the earlier research of Lewis, to be a necessary prelude to conducting quantitative research. That is, we regard our work as the identification of interesting aspects of a novice programmer, which may then be studied quantitatively, either by us or by other researchers.

2 Week 3: Tracing Code

Each think aloud session with Donald was recorded with a Livescribe Smartpen (2013) which captured everything that Donald wrote and spoke. The scripts that Donald completed were then processed to produce "pencast" PDFs, the audio-synced video contents of which are re-playable using Adobe Acrobat Reader. The audio was also transcribed. Ellipses ("...") are used throughout the transcripts to indicate both missing utterances which add little to the context (for example, sighs, laughs, coughs, and fillers such as "um", "mmmm", and "huh"), and also short pauses in articulation.

The first think aloud task performed by Donald is shown in Figure 2. Donald performed this task in week 3 of semester, but he had already discussed this problem with his lecturer, at a one-on-one meeting. The lecturer had shown Donald a way to perform a systematic trace on that code.

Donald began by writing out the code as shown in Figure 3. The left hand sides of lines 1 to 5 and also lines 6 to 10 are the code from Figure 2. The right hand sides of lines 1 to 5 were subsequently written by Donald as he updated variables during his trace. As we shall see, writing those updated values on the right hand side may be one source of his subsequent confusion during the trace.

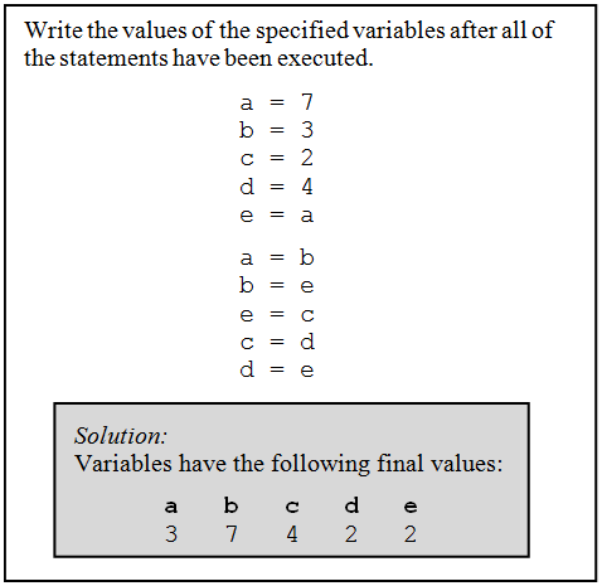

Figure 2: Donald's Week 3 Tracing Task

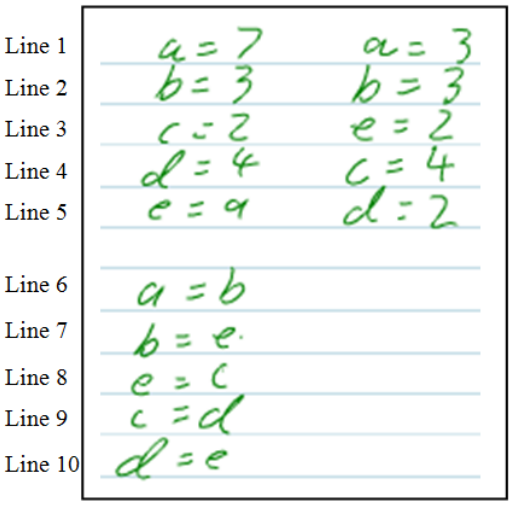

Figure 3: Donald's attempt at his Week 3 Tracing Task

As he began his trace, Donald recalled that his lecturer had used a systematic way to record a trace:

> *I remember there was an easier way to do this, visually … a way to write this out to make it very easy to … represent.*

He then started tracing the code from line 6, writing the new values stored in each variable next to the first block of given code (i.e., lines 1 to 5). This was NOT the layout that the lecturer had demonstrated to Donald. Note that, in starting at line 6, Donald ignored line 5. As he wrote "`a = 3`" on the right hand side of line 1, Donald said:

> *So immediately, well if* `a` *equals* `b`*,* `a` *equals 3*

He then looked at line 7 and said:

> *b equals e. … ah … that change it? No. … b = 3, so b right now equals a, which equals now 3.*

From the transcript of the think aloud session, it is not clear what Donald meant by "*that change it?*" However, a year after Donald performed this think aloud, we had him listen to this pencast, and he explained that at that stage of his development he had been unsure of what assignment statements did:

> *My confusion with* [for example] *e = a was due to me not realising that e received a copy of the value of a and then they are separate. I thought they were still connected.*

That novice misconception about assignment statements – that it "connects" variables – is well known (e.g., Du Boulay 1989). As a consequence of that misconception, Donald incorrectly wrote on the right of line 2, "`b = 3`". That is, he reasoned (incorrectly) that `e` had been "connected" to `a` at line 5, then `a` had been "connected" to `b` at line 6, so at line 7 `b` was effectively being set to itself – hence his "no" in response to his own question "*that change it?*".

Donald then correctly evaluated lines 8 and 9, writing that "`e = 2`" and "`c = 4`". At line 10, he began correctly:

> *… c equals d … which is 4, and d equals e … which equals 2*

But then Donald added:

> *… if I go to the updated version*

Donald was not sure he should be using the "updated" value "`e = 2`" as he wrote on the right hand side at line 3, or whether he should follow a similar chain of reasoning as before – that line 8 "connected" variables `e` and `c` and line 9 connected `c` to `d`. To follow that chain of reasoning would be consistent with how Donald (incorrectly) evaluated line 7, but in writing "`d = 2`" on the right hand side he elects to *not* be consistent. Donald then revisited his trace, to check for errors:

> *So just to revise … a starts off equalling 7. However it needs to equal b which equals 3. … b equals 3. Hang on, but b equals e … which equals a, which equals 3. Yeah … ok … e … Oh! I totally missed that …*

By "*I totally missed that",* Donald was indicating that he had not previously noticed line 5, "`e = a`":

> *e equals … a, which equalled 3. …* [sound of air being sucked through teeth] *… Yeah, because e has already been changed.*

Having become confused, Donald restarted his trace from the beginning. The following extract from the transcript may not be coherent, but it does illustrate his growing confusion:

> *a equals b, and b equalled 3 … b equals e, and e equalled a, which then became 3 … so then e equals c … e equals … a…arrgh….so then c becomes 3. … No it doesn't. Does it? Arrgh!*

But Donald showed determination. He started another review of his trace, and when he became confused, he restarted yet again. He began that trace by reciting the first four lines of code:

> *… a has been assigned 7, b has been assigned 3, c has been assigned 2, d has been assigned 4.*

For these four lines, his choice of the word "assigned" was a change from his earlier use of "equals". He then articulated line 5 differently, using the word "equal":

> *... and e has been said to be equal to a, which is 7. So it's currently 7 ....*

Thus in reciting lines 1 to 5 he articulated (a) the correct conception that a constant value to the right of an equals sign indicates an assignment of that constant to the variable on the left hand side, but (b) the misconception that variables on both sides of an equals sign "connects" the two variables.

Donald then continued on to correctly evaluate line 6, but at line 7 his misconception about line 5 lead him astray:

> *... b now has the value of e, e has the value of a, and I changed the value of a, so that makes it have the value of also 3.*

At this point, Donald had been working on this problem for 7 minutes and 18 seconds. He continued for another two minutes, while becoming even more confused, before writing his final answer. Donald expressed low confidence in his final answer:

> *I still think that is wrong. I'm not really sure. I think it's sort of right.*

In fact, Donald's answer happened to be correct for all the variables except `b`, for which he had the value 3 instead of the correct value of 7. However, had Donald consistently applied his misconception – that variables on both sides of an equals sign "connects" the two variables – then lines 5 to 10 should have "connected" all the variables, in which case all the variables would then have the same value. Near the end of his trace, Donald actually made that same point:

> *... I thinking I'm just changing everything to 3 now by accident but we'll see what happens.*

While Donald's misconception about assignment statements has been well known for decades, what we see in Donald's think aloud is that he does not apply that misconception consistently. According to neo-Piagetian theory, such inconsistency is common in novices reasoning at the sensorimotor stage.

Later in this think aloud session, Donald reflected on his *ad hoc* approach to recording his trace, especially his recording of variable values on the right hand side of lines 1 to 5:

> *... I represented it the wrong way. I probably should have had this be more ... like ... move it down so it is in a line ... rather than try to do this and then go back up.*

When we interviewed Donald a year after he had done this think aloud session, he reflected on the problems he was having at the time he did this trace:

> *... it takes me a very long time to remember how to think like a computer, and that's really what I find slows me down, because my mind wants to try and handle it a different way – but I'm like "No, a computer! You go line by line" ... but to me that's not the first way my mind wants to work ... I don't have that automatic ... a computer is very simple actually. Looks like it is very complicated ..."*

In summary, at this week 3 think aloud, Donald displayed the characteristics of a programming novice working at the sensorimotor stage. The misconceptions he had about programming concepts were applied inconsistently. He was cognitively overloaded on a simple tracing task as he was unable to organise an effective and accurate method for tracing code.

In classical Piagetian theory, the sensorimotor stage is experienced by infants. In the application of neo-Piagetian theory to novice programmers, the use of the term "sensorimotor" to describe the initial stage remains appropriate, since at this stage the novice programmer has trouble interpreting the semantics of the code he or she is reading (i.e. the sensory component) and also has trouble with writing down a well organised trace (i.e. the motor part). Furthermore, the sensory and motor components interact. For example a misconception about what a piece of code does can lead to an incorrect method of recording within a trace the result of applying that misconception.

3 Week 9: Explaining By Tracing

At his week 9 think aloud session, which was his fifth such session, Donald attempted the explanation problem in Figure 4. We have already described this particular think aloud by Donald in an earlier publication (Teague *et al*. 2013). Here we summarise those aspects of the think aloud that are most salient to this paper.

Donald attempted to explain the code by using the inductive approach of a novice at the preoperational stage of neo-Piagetian theory. That is, he selected some initial values for the variables (`y1 = 1`, `y2 = 2` and `y3 = 3`), then traced the code with those values, and then inferred what the code did from the input/output behaviour. However, at week 9, Donald was still having some problems organising his trace, so his use of the inductive approach did not initially go smoothly.

Donald began with the unsuccessful trace shown in Figure 5. Like his week 3 trace, this trace was not well organised. Each of the three lines of that trace represents an `if-then` block from the code in Figure 4. The numbers and arrows are Donald's attempt to record how the values in the variables change as the code is executed. Unlike his week 3 trace, the transcript of this week 9 trace shows that Donald had a correct and consistent understanding of how the code works. But by the time he reached the third line of that trace, his method of recording the values led him to confusion.

Donald then attempted a second, more organised trace, as shown in Figure 6. He first wrote, on each of the three lines, respectively "`y1 = 1`", "`y2 = 2`" and "`y3 = 3`" (the numbers "`1`", "`2`" and "`3`" were subsequently crossed out as his trace progressed). He then performed a conventional and correct trace, which took him only 67 seconds. In performing this second trace, Donald showed clear progress from the haphazard sensorimotor approach he used in week 3.

However, based on this one successful trace, Donald then made an incorrect inductive inference, which led him to write the following incorrect answer:

> *"To reverse the values stored in `y1`, `y2` and `y3` ..."*

We then asked Donald to trace the code again, using the initial values y1 = 2, y2 = 1 and y3 = 3. He performed a

successful trace with those values, using the same approach as in his previous trace. On completing this trace with our values, however, Donald initially maintained that this trace confirmed his initial answer, with this trace having "*ended up the same ... as what I originally came up with*". (Although his tone of voice in the recording might indicate uncertainty, or irony.) After being challenged by us, but without us providing any further hints, Donald exclaimed:

> *"Oh! It's ordering them ... um ... so, it's more about, it's not to rev ... hang on ... oh* [indecipherable]*… rather than to reverse, it would be to, place them from highest to lowest."*

If you were asked to describe the purpose of the code below, a good answer would be "*It prints the smaller of the two values stored in the variables* `a` *and* `b`".

```
if (a < b):
    print a
else:
    print b
```

In one sentence that you should write in the empty box below, describe the purpose of the following code.

Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code, like the purpose given for the code in the above example (i.e. "*It prints the smaller of the two values stored in the variables* `a` *and* `b`").

Assume that the variables y1, y2, and y3 are all variables with integer values.

In each of the three boxes that contain sentences beginning with "`Code to swap the values …`" assume that appropriate code is provided instead of the box – do **NOT** write that code.

```
if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
if (y2 < y3):
    Code to swap the values in y2
    and y3 goes here.
if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
```

Sample answer:
It sorts the values so that y1 ≥ y2 ≥ y3

Figure 4: The Week 9 Explain in Plain English Task

In this week 9 think aloud, Donald initially showed attributes of the sensorimotor stage, but he then went on to also show some of the attributes of the preoperational stage. After an initial unsuccessful trace, he performed two well organised and successful traces. However, using the inductive approach based on the input/output behaviour of his first successful trace, Donald jumped to a rash and incorrect answer. This answer was especially rash, because the initial values he chose resulted in all the `if` conditions being true. (Some of the other students who participated in our think aloud study did carry out an initial trace with the same values chosen by Donald, but they also carried out a second trace with different values.) However, when Donald was prompted to perform a second trace, with values given to him by us, he did infer a correct description of the purpose of the code. Donald manifested behaviour consistent with someone who, in terms of the overlapping wave metaphor, is transitioning from the sensorimotor stage being dominant to the preoperational stage being dominant. In this week 9 think aloud, Donald did not manifest any aspect of concrete operational reasoning.

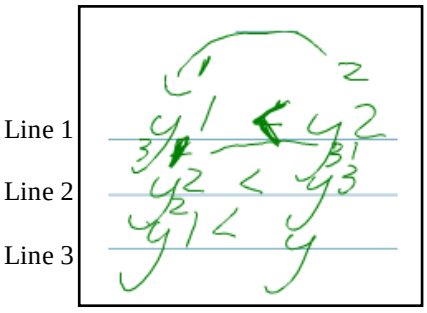

Figure 5: Donald's First Week 9 Trace.

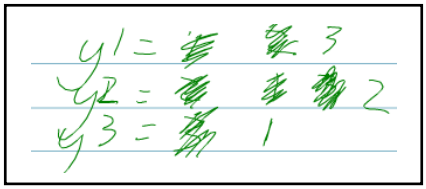

Figure 6: Donald's Second Week 9 Trace

3.1 The Concrete Operational Approach

Instead of reasoning about the Figure 4 problem in terms of specific values, as Donald did, a novice reasoning at the concrete operational stage would tend to reason (albeit implicitly) about the code in terms of algebraic constraints on the values in the variables. For example, after the first `if` statement in the code from Figure 4, the concrete operational novice would think of `y2` as holding any possible value that satisfies the condition that it is less than the value in `y1`. After the second `if`, the concrete operational novice thinks of `y3` as holding any possible value that satisfies the condition that it is less than the values in both `y1` and `y2`. By thinking in this deductive fashion, the concrete operational student feels little need to understand code via the inductive, preoperational approach of tracing specific values.

4 Week 13: Abstract Reasoning

In neo-Piagetian theory, one of the defining characteristics of the concrete operational novice is the ability to reason about abstract quantities that are conserved. For example, in a classic Piagetian experiment, a preoperational child believes that when water is poured from one container into another, and the

water level is higher in the second container, then there is more water in the second container than there was in the first. In contrast, an older child at the concrete operational stage is aware that the quantity of water is conserved.

In a programming context, Lister (2011) conjectured that a preoperational programming student would tend to think that small changes to the implementation of an algorithm would change the specification of what the code does. Equally, Lister argued that a concrete operational student should be able to make small implementation changes to code while conserving the specification. He nominated a problem like that shown in Figure 7 as an example of a problem that requires a concrete operational understanding of programming. We had Donald attempt this problem at his week 13 think aloud session.

Figure 8 shows Donald's attempt at this week 13 task. (Note that Donald wrote his answers for this task on a blank page. We have superimposed his answers over the question text in Figure 8, and retained the sample answers in boxes on the right of Figure 8, to make it easier for the reader to follow.)

In the three boxes in Figure 8, Donald only provided correct code for one box. However, neo-Piagetian theory merely says that a student progresses from sensorimotor, to preoperational to concrete operational *when the programming constructs to which the novice is exposed do not change*. When new programming constructs are introduced (as loops and arrays are in the week 13 problem) then a novice may need to pass through the neo-Piagetian stages for these new constructs. Thus, a student may have a concrete operational grasp of non-iterative and non-array aspects of a piece of code, but at the same time be reasoning about the iterative/array aspects at the preoperational or sensorimotor stage. On inspection of the transcript for Donald's attempt at the week 13 problem, it is obvious that he struggled with the distinction between a *position* in an array and the *contents* of that position – as many novices do when they first encounter arrays. Therefore, with respect to arrays, especially when writing code, Donald is at the sensorimotor stage of development.

However, a close inspection of the transcript shows that Donald has made some progress since week 9 with reasoning about other code. The remainder of this section will emphasise the transcript evidence for the progression in aspects of Donald's reasoning.

As Donald began by reading the problem, he articulated a clear understanding of what was required, and a clear acceptance that two different implementations might satisfy the same specification:

> *So it does the same thing, but is going to be doing slightly different code because … line 5 is different.*

As Donald read through listing 1, he articulated an imprecise description of line 2, indicative of his weakness in distinguishing between a position in an array and its contents:

> *… x zero is best …*

In the **Source Code Listing 1** below is code for a function which returns the smallest value in the array `x`. When the code in **Source Code Listing 2** below is correctly completed, it should also return the smallest value in the array `x`. Line 5 is different in the two listings. Except for line 5, and the lines with boxes, all other lines in the two listings are the same

Complete the code in the boxes in the second listing on lines 2, 4, and 8 so that the method `Min` returns the smallest value in the array `x`.

Source Code Listing 1

```
1.  public int Min(int[] x) {
2.    int best = x[0];
3.    for (int i=1; i<x.Length; i++){
4.     if (x[i]<best) {
5.        best=x[i]; // different from line 5
6.     }             // in the second listing
7.    }
8.    return best;
```

Source Code Listing 2

```
1.  public int Min(int[] x) {
2.    int best = [ 0 ] ;
3.    for (int i=1; i<x.Length; i++){
4.     if (x[i] < [ x[best] ] ) {
5.        best = i;   // different from line 5
6.     }              // in the first listing
7.    }
8.    return [ x[best] ] ;
```

Figure 7: The Week 13 Task & sample answer

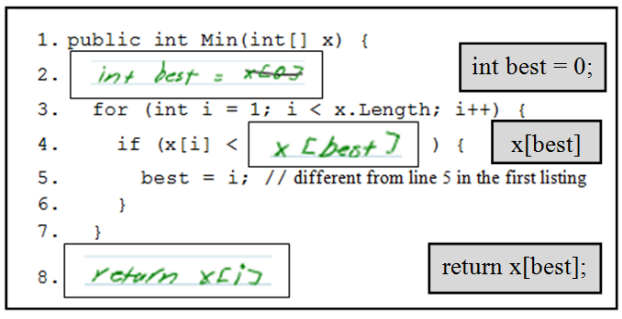

Figure 8: Donald's Attempt at the Week 13 Task.

After reading lines 3, 4 and 5 of listing 1, Donald then summarises the entire loop in a way that shows some nascent signs of concrete operational reasoning:

Ok, so it's just going through the list … so every time it finds something smaller it assigns to best until we get to the end …

Had Donald then added something like "*so at the end of the loop best will contain the smallest value in the array*" that would have been unambiguous evidence for concrete operational reasoning, but what he actually uttered is at least a coherent summary of the four lines of code that form the loop.

Donald then read listing 2. He briefly adopted a quasi-preoperational approach to reasoning about that code, by considering how the code in the second listing would work for the specific case of the first iteration of the loop:

If x at position 1 is less than x at position 0, it would take the element number of i … and then assign it to best. Then element one … has the least.

Note, however, that while Donald considered two specific *positions* in the array, he did not consider specific *values* at those positions. Nor does he consider any other specific positions in the array. He appears satisfied that his consideration of positions 0 and 1 is representative of what will happen for subsequent iterations of the loop. Again, this is an example of nascent concrete operational reasoning – he is not performing a *complete* trace with *specific* values, as he did in week 9.

Donald then started writing his answer. After writing "`int best = `" in the first box, he hesitated and then had a stroke of insight about the third box:

… return x i. Ah! … I think I got it!

He then wrote his (incorrect) answer in the third box, before completing his incorrect answer in the first box. (The line through `x[0]` in the first box is not relevant and should be ignored.) In his incorrect solution for box 1, Donald displayed his sensorimotor difficulty in distinguishing between the position in an array and the contents of that position, but in the way that he worked on the first and third boxes simultaneously, he does at least demonstrate some concrete operational grasp of the relationship between the code in those two boxes.

Donald then wrote his correct answer in the middle box, while again articulating a quasi-preoperational justification for his answer. In so doing he again connects the code in two of the boxes, this time the first and second box:

… So if element 1 is less than best, and we start best off at 0, then … it would become 1 … I think that would work.

Donald then voluntarily checked his solution by completing a trace. For the array values, he chose 2, 1 and 3, in that order. In placing those array values in that order, Donald demonstrated a more sophisticated choice of initial values than he had for the week 9 problem. In his subsequent trace, he arrived at the wrong answer because of his sensorimotor difficulty in distinguishing between the position in an array and the contents of that position. (His incorrect trace also reinforced his belief that his solution was correct)

It had taken Donald about nine and a half minutes to complete this exercise. In the subsequent debrief with us, most of the discussion centred on his sensorimotor difficulty in distinguishing between the position in an array and the contents of that position. After we had helped him correct his answer for the first box, he immediately corrected the third box without any help from us. In so doing, he showed some nascent concrete operational understanding of the relationship between the code required in the first and third boxes.

In summary, Donald's weakness with arrays was obvious in this week 13 task. However, if his weakness with arrays is ignored, then there are signs in the week 13 task that he had begun to progress beyond the exclusively inductive approach he used in week 9. That is, he showed some capacity to reason about code without needing to perform a complete trace with specific values.

5 Concrete Reasoning

Figure 9 shows Donald's attempt at the problem in Figure 7 three semesters after his attempt shown in Figure 8. He was by then nearing completion of his degree and had successfully completed six programming courses. Donald approached the task with confidence:

… should be in principle pretty easy to do. So if I look at the first code public int min, so pass in the array … then we just iterate through incrementally … and if the current is less than best, we pass that in.

While Donald did, in the above transcript extract, articulate three keywords (i.e. public int min) as he often did in earlier think alouds, here he went on to articulate an abstraction beyond just the keywords, for example:

so pass in the array

instead of "int x";

then we just iterate through incrementally

instead of articulating the lexical symbols on line 3;

and if the current is less than best

instead of articulating the lexical symbols on line 4, with his use of "current" suggesting an abstraction beyond the code itself, which is consistent with a subsequent articulation of the `for` loop at line 3:

… … int i … is assigned 1 and then it keeps going through … the length of the array

Donald has given a reasonable explanation for the functioning of the `for` loop. He has done this in abstract terms, rather than relying on specific values of elements or indexes to explain what's going on. Earlier in his development, as described in the previous section, Donald's behaviour had been more pre-operational, and he had relied on specific index positions when he talked about the same looping structure (i.e., *if x at position 1 is less than x at position 0, it would take the element number of i*). Donald now further demonstrates that he has developed an ability to explain code in an abstract manner:

so if the current element of x is less than best, i … which is the value of that, is put into best.

In the above transcript extract, it is unclear whether Donald is thinking of the variable `best` as being a value copied from the array, or `best` as representing a position in the array. However, he begins to improve the clarity of

his thinking when he focuses on line 5 in the second listing, and how it differs from line 5 in the first listing:

> *... it's putting i into best and that's ... why would that be a problem?*

Donald re-reads the question and then articulates a clear distinction between the contents of an array position and the position itself:

> ... [paraphrasing the question text] *"it should return the smallest value in the array x" ... Ah!* [Whereas] *I return the index of where the smallest ... value is ... in the array.*

Donald then goes on to produce a correct answer to this task, with no hesitation or backtracking, which is shown in Figure 9. As he does so, Donald says the following:

> *So we start with um, 0. Yep. ... and then x at index best ... and then we return ... x index best.*

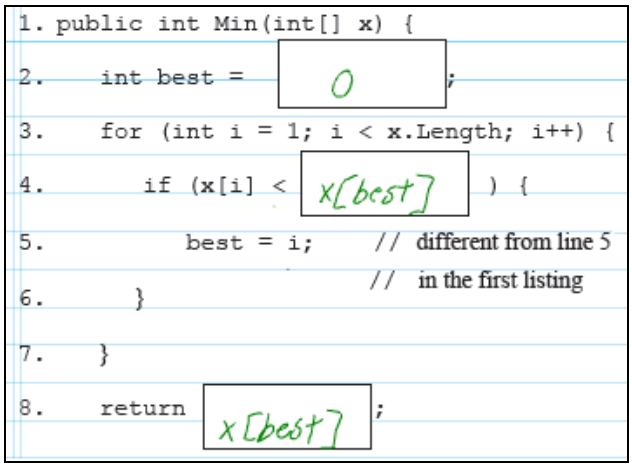

Figure 9: Donald's attempt at the Figure 7 task three semesters later.

In this think aloud session, near the end of Donald's undergraduate studies, he demonstrates a much improved ability to reason in terms of abstractions beyond the code itself, compared to his earlier attempt at this same exercise. His ability to reason abstractly, consider consequences, and complete the task quickly and accurately provides solid evidence of his progression into the concrete operational stage.

Also, given Donald's close reading of the given code, and his initial confusion over what his code should be returning (i.e. a value from the array or a position from the array), it is clear that Donald remembered little of his first attempt at this problem when he did it this second time.

6 Conclusion

Across the sequence of think aloud sessions presented in this paper, Donald manifested developmental stages consistent with neo-Piagetian stage theory. First we witnessed him performing at the sensorimotor stage: using considerable cognitive effort to trace simple code; unable to trace reliably and accurately. We saw him gradually develop skills consistent with the next stage of preoperational: tracing code more reliably, but still being unable to reason deductively about code or see a relationship between different parts. Then finally we saw evidence of his transition into the concrete operational stage where he can reason and explain the purpose of code, talk in terms of abstractions rather than specifics and consider consequences and alternatives.

At this stage of our research programme, the question remains as to whether Donald represents a significant portion of novice programmers. Based upon our work with other students, we suspect he is not a rare exception, but that will need to be confirmed by quantitative research.

The computing community has tried many variations on how to teach programming, but many students continue to struggle. Neo-Piagetian theory points to one aspect of programming pedagogy that has remained largely invariant across those many past variations – our teaching skips too quickly across the sensorimotor and preoperational stages for many students. We suggest that teaching be designed explicitly with students' current level of reasoning in mind. As the cognitive skills developed through the neo-Piagetian stages are sequential and cumulative, novices need to be reasonably strong at a lower neo-Piagetian stage before they can be expected to reason well at a higher neo-Piagetian stage. Otherwise, teachers are in danger of talking to their students in a way that the students are not yet capable of processing.

Some computing academics claim that students who struggle to learn programming lack an innate talent for programming. Any readers of this paper who share that suspicion might think that Donald's early performance in think alouds indicated that he lacked such a talent. Those readers may be surprised to learn that Donald has completed his degree with a high grade point average (more than 6 out of a possible 7), and is, in the near future, taking up a fulltime position at an international corporate professional services firm as a business IT consultant. Donald's academic achievements may indicate that programming ability is something that is learned, rather than something innate. Donald remained enthusiastic and determined no matter how hard he found the tasks we gave him. He saw those tasks as learning experiences, and consequently he improved. Perhaps Donald personifies the primary qualities required to learn programming – perseverance, a desire to learn – not an innate ability to program. What instructors need to do is provide instruction targeted at an appropriate level of abstract reasoning for their student(s), rather than assume that students have the cognitive maturity to perform programming tasks requiring concrete operational reasoning.

7 Acknowledgments

Our heartfelt thanks and congratulations to Donald, whose approach to learning is an inspiration to us. Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office for Learning and Teaching or the Australian Government.

8 References

Bennedsen, J. and Caspersen, M. (2008): Abstraction ability as an indicator of success for learning computing science? Proc. of *4th International Workshop on Computing Education Research* (ICER 2008) Sydney, Australia, 15-26.

Biggs, J. and Collis, K. (1982): *Evaluating the quality of learning: The SOLO taxonomy* (*Structure of the Observed Learning Outcome*). Academic Press, New York, NY.

Bloom, B.S., et al. (1956): T*axonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company.

Bond, T. (2005): *Piaget and the Pendulum.* In Matthews, M., Gauld, C. and Arthur Stinner, A. (Eds) *The pendulum: scientific, historical, philosophical and educational perspectives.* Springer. 303-313.

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

Cole, M. (1996): *Cultural Psychology: A Once and Future Discipline*. Cambridge, Mass: Belknap Press of Harvard University Press

Corney, M., Teague, D., Ahadi, A. and Lister, R. (2012): *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Proc. of *14th Australasian Computing Education Conference* (ACE2012), Melbourne, Australia. **123**:77-86, ACS.

Demetriou, A., Shayer, M. and Efklides, A. (Eds). (1992): *Neo-Piagetian Theories of Cognitive Development: Implications and Applications for Education*. London : New York: Routledge. ISBN 0415117496.

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

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

Flavell, J., Miller, P. and Miller, S. (2001): *Cognitive Development* (4th Edition). Pearson. ISBN: 0137915756.

Inhelder, B. and Piaget, J. (1958): *The Growth of Logical Thinking from Childhood to Adolescence*. Routledge & Kegan Paul, London.

Lister, R., Clear, T., Simon, Bouvier. D., Carter, P., Eckerdal, A., Jackova, J., Lopez, M., McCartney, R., Robbins, P., Seppala, O. and Thompson, E. (2010): Naturally occurring data as research instrument: analysing examination responses to study the novice programmer. *SIGCSE Bull.* 41, 4 (January), 156-173. http://doi.acm.org/10.1145/1709424.1709460

Lourenco, O. and Machado, A. (1996): In defense of Piaget's theory: a reply to 10 common criticisms. *Psychological Review* 103, **1**:143-164.

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

Levin, I. (Ed.) (1986): *Stage and Structure: Reopening the Debate*. Norwood, New Jersey: Ablex.

Lewis, C. (2012): *The importance of students' attention to program state: a case study of debugging behavior*. In Proc. of *9th Annual International Conference on International Computing Education Research* (ICER 2012). Auckland, New Zealand, September 10 – 12. 127-134.

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, Australia. 9-18. http://crpit.com/confpapers/CRPITV114Lister.pdf

LiveScribe. (2013): http://www.smartpen.com.au/ Accessed April 2013.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001): A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. SIGCSE Bull., 33(4). 125-140.

Siegler, R. (1996): *Emerging Minds: The Process of Change in Children's Thinking*. New York: Oxford University Press.

Siegler, R. S. (2006): *Microgenetic analyses of learning*. In W. Damon & R. M. Lerner (Series Eds.) & D. Kuhn & R. S. Siegler (Vol. Eds.) *Handbook of Child Psychology, Vol. 2: Cognition, Perception and Language* (6th ed) pp. 464-510). Hoboken, NJ: Wiley.

Sutherland, P. (1992): *Cognitive Development Today: Piaget and his Critics*. London: Chapman. ISBN 1853961337.

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* (ACE2013), Adelaide, Australia. CRPIT Vol. 136. Carbone, A. and Whalley, J., Eds. 87-96. http://crpit.com/Vol136.html

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P. and Prasad, C (2006): *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies*. Australian Computer Science Communications 52: 243-252. http://crpit.com/confpapers/CRPITV52Whalley.pdf