Manifestations of Preoperational Reasoning on Similar Programming Tasks

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

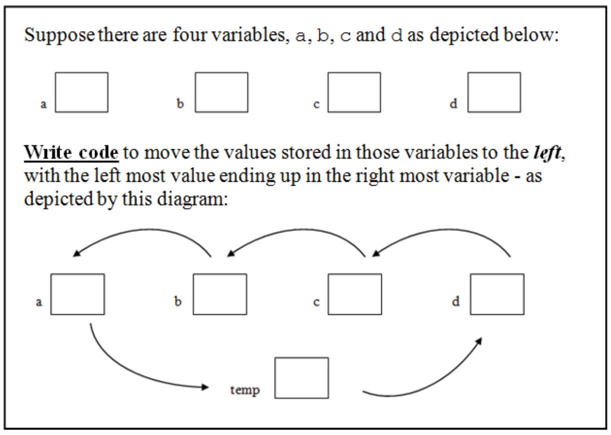
Abstract

In this research paper, we study a simple programming problem that only requires knowledge of variables and assignment statements, and yet we found that some early novice programmers had difficulty solving the problem. We also present data from think aloud studies which demonstrate the nature of those difficulties. We interpret our data within a neo-Piagetian framework which describes cognitive developmental stages through which students pass as they learn to program. We describe in detail think aloud sessions with novices who reason at the neo-Piagetian preoperational level. Those students exhibit two problems. First, they focus on very small parts of the code and lose sight of the "big picture". Second, they are prone to focus on superficial aspects of the task that are not functionally central to the solution. It is not until the transition into the concrete operational stage that decentration of focus occurs, and they have the cognitive ability to reason about abstract quantities that are conserved, and are equipped to adapt skills to closely related tasks. Our results, and the neo-Piagetian framework on which they are based, suggest that changes are necessary in teaching practice to better support novices who have not reached the concrete operational stage.

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

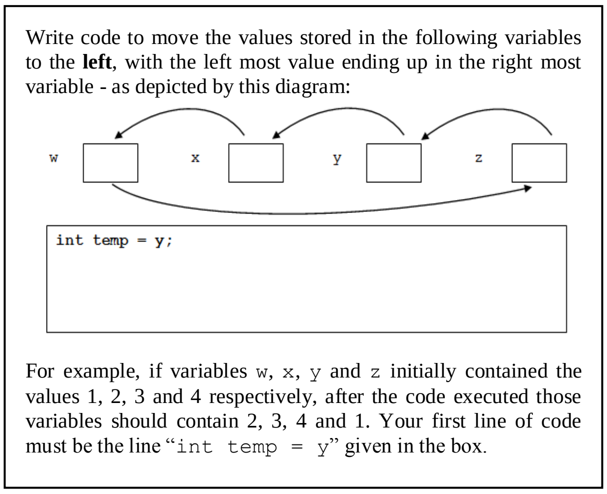
1 Introduction

It is a common source of frustration for computer science educators that novices do not transfer to a second programming problem the concepts taught on an initial problem. For example, we posed to novice programmers the tasks shown in Figures 1 and 2. We found that some students who could do the first task could not do the second task. We posed these questions to two classes, in different semesters. Table 1 shows the performance of both classes on the second task. In both semesters, the percentage of students who answered the second task incorrectly was worse than we expected, given the number of weeks of instruction the students had received.


Suppose there are four variables, `a`, `b`, `c` and `d` as depicted below:

Write code to move the values stored in those variables to the ***left***, with the left most value ending up in the right most variable - as depicted by this diagram:

Figure 1: The shift task with an explicit temp variable


Write code to move the values stored in the following variables to the **left**, with the left most value ending up in the right most variable - as depicted by this diagram:

```
int temp = y;
```

For example, if variables `w`, `x`, `y` and `z` initially contained the values 1, 2, 3 and 4 respectively, after the code executed those variables should contain 2, 3, 4 and 1. Your first line of code must be the line "`int temp = y`" given in the box.

Figure 2: The second shift task

Week of Semester	No. of Students	Percentage wrong
10	51	37%
6	113	53%

Table 1: Performance on the Task in Figure 2

To understand why so many students struggled with such a simple task, we began the qualitative research study described in this paper. In our study, we had 11 volunteer students complete the tasks in Figure 1 and 2, while having those students think aloud as they did so.

Table 2 summarises the performance of the 11 students. The names shown in that table are all pseudonyms. All of these students were in at least their third week of learning to program. All 11 students completed the first task successfully. In completing that first task, those 11 students demonstrated that they understood assignment statements, and that they understood the English language instructions associated with both tasks. However, 3 of the 11 students could not then solve the second task, and a fourth student (Jim) took much longer. (Those four students are in the shaded region of Table 2.) This brings us to the research question addressed in this paper:

Research Question: *Why can some students answer correctly only one of the two problems shown in* Figures 1 and 2, *when both tasks require functionally identical code*?

Note that our research question is not related to the prevalence of this issue in the general population of programming novices. Given the small group of students we studied, and that those students are from a single institution, it would not be appropriate to speculate on prevalence. However, what we can do in a qualitative study of this type is arrive at a possible explanation for why some students find the second task to be significantly harder than the first task. The type of micro-genetic analysis that we carry out in this study has been applied in many domains to test theories of cognitive development (Siegler 2006) and has also been used before in a study of novice programmers (Lewis 2012).

We were able to make sense of our research data via neo-Piagetian theory. In the next section, we briefly describe that theory. We then present our transcript data from three students, two of whom struggled on the second task while the third student was able to do both problems quickly. We interpret that transcript data using the neo-Piagetian theoretical framework.

2 The Neo-Piagetian Stages

Lister (2011) proposed, in accordance with neo-Piagetian theory, that there are four main stages of cognitive development in the novice programmer. At the least mature stage, the sensorimotor stage, a novice programmer cannot reliably trace a given piece of code (i.e., manually execute it). The sensorimotor approach to writing a trace on paper is ad hoc and often inconsistent. Also, they commonly have misconceptions about what various programming constructs do (Du Boulay 1989). Furthermore, these novices often apply a misconception at some points in a trace and then apply a correct conception at other times.

The next neo-Piagetian stage is preoperational. Novices at this stage can trace code accurately, but they struggle to reason about code. That is, they have difficulty understanding how several lines of code work together to perform a computation. At any point in time, these novices tend to be focused on small parts of the code, and ignore the implications of code they have already considered. This is what neo-Piagetian theorists refer to as *spatial and temporal centration.*

At the concrete operational stage, novices can reason with abstractions of code (e.g., diagrams). They can also reason about the concept of conservation which Flavell (1977) describes as "… *a quantitative invariant amid transformations*". We elaborate on the concept of conservation in the following sub-section.

Finally, there is the formal operational stage, which is the stage educators hope their students will reach. At this stage, novices can reliably and efficiently "problem-solve"; they understand and use abstractions, form hypotheses and can make inductive and deductive inferences.

By analysing students' answers in an end-of-semester exam, Corney et al. (2012) provided indirect evidence that novices pass through some of these neo-Piagetian stages. However, such evidence does not provide a direct indication of the actual thought processes of students. Think aloud studies have also been undertaken with students who were given programming code to hand trace and explain in plain English (Teague, Corney, Ahadi, and Lister 2013). The results provided evidence of preoperational reasoning by some of the students.

In this paper we provide direct empirical evidence of students' thought processes while solving code writing tasks, specifically the tasks shown in Figure 1 and 2.

2.1 The Concept of Conservation

According to neo-Piagetian theory, it is only at the concrete operational stage that a novice has developed the ability to reason reliably about abstract quantities that are conserved, and the novice is not deceived by superficial appearances. For example, Flavell (1977) describes an experiment where a preoperational child believes that when clay is moulded into different shapes the amount of clay changes. A child at the concrete operational stage is not deceived by such perceptions. Lister (2011) proposed that in a programming context, a novice at the concrete operational stage should be able to easily make minor changes to code while conserving what the code achieves, while the preoperational novice programmer would struggle to do the same. The contribution of this paper is providing empirical evidence for that proposal.

Our objective was to see if any of our novices demonstrated an ability to conserve a specification when given a small change to the implementation. Specifically, we wanted to see if any of our novices could solve either the first or second task, but not both. Our hypothesis was that students who are operating at the preoperational level will struggle to apply consistently the abstract principal common to both problems – that saving a variable to `temp` makes it possible to overwrite that value in the copied variable. In neo-Piagetian terminology, this abstraction is referred to as the "*invariant amid transformations*" (Flavell 1977).

2.2 Working with Cyclic Series

Our two programming tasks are analogous to an experiment Piaget conducted where he asked children to predict the next element in a cyclic series (Piaget 1971a). To do so required the children to translate the elements into a linear series. Piaget found that relationships of order are operational. That is, people are not capable of dealing with such a concept until the concrete stage.

Alias	The First Shift Task (see Figure 1)		The Second Shift Task (see Figure 2)		
	Time (minutes:seconds)	Help Given	Weeks after first think aloud	Time (minutes:seconds)	Help Given
John	2:03	0. none	4	1:04	0. none
Steve	1:48	1. clarify	3	1:12	0. none
Becki	1:05	0. none	0	2:40	0. none
Michael	1:24	1. clarify	0	2:30	0. none
Bobcat	14:36	3. hint	0	2:40	2. prompt
Lance	3:10	0. none	7	2:40	0. none
Johnstone	*4:48*	*3.* hint	2	*2:51*	0. none
Donald	*3:44*	2. prompt	0	*8:49*	2. prompt
Charlotte	*7:45*	3. hint	0	*10:00*	4. provide
Potato Man	*19:02*	3. hint	3	*17:30*	4. provide
Jim	*5:43*	1. clarify	2	*21:37*	4. provide

Table 2: Think Aloud Performance on Shift Tasks

At the sensorimotor stage, people are barely able to manage translating a cyclic series into a linear series and unable to foresee successive elements. At the preoperational stage people have the ability to predict successive elements in a cyclic series *iff they start at the first element*. Towards the end of the preoperational stage, people can cope with intermediate starting points, but still fail to predict elements beyond the last.

Our programming tasks involved transforming a cyclic series (the diagram) into a linear series of assignment statements to achieve a 'movement' of values.

3 Think Aloud Results

At some point in time after performing a think aloud on the first task, the 11 students performed a think aloud on the second task. The elapsed time between think alouds varied from student to student. Table 2 provides the specific information for each student.

Table 2 also shows the total time taken to complete (or abandon) each task. The data in Table 2 is sorted by length of time spent on the *second* task. Thus the four students at the bottom of Table 2 (i.e. in the more heavily shaded section of the table) took the longest time to complete the second task. According to the arguments we have made above, those four students are likely to be at the preoperational level of development.

Table 2 also shows the level of assistance provided to each student by the person conducting the think aloud. We have categorised that level of assistance using a scale adapted from Perkins & Martin (1986):

0. *none*	No intervention by interviewer.
1. *clarify*	Clarification of the task requirements (e.g., explaining terminology in task text).
2. *prompt*	Prompting to encourage progress (e.g., reflecting on what has been done so far and asking what needs to happen next; intimating there may be an issue; or suggesting that they manually execute the code).
3. hint	Hinting in order to provide some direction (e.g., suggesting a programming construct or indicating where an issue lies).
4. *provide*	Providing a partial or complete solution if progress seems unlikely; or the subject has abandoned the task.

4 Dissection of Think Alouds

In this section, we dissect the think aloud sessions of Charlotte, Jim and Steve. Because of space limitations, we are unable to include the entire transcript for these students, and we have therefore chosen a selection of short excerpts which are representative of their attempts. Charlotte and Jim are typical of all four students who could solve the first task, but struggled with the second. Our presentation of each excerpt is broken into three subsections (summary, data, and analysis), following the format used by Lewis (2012).

4.1 Charlotte

Charlotte was in her third week of learning to program when she performed the following think aloud. This was her second think aloud session, and she was comfortable with the protocol of articulating her thoughts as she solved programming tasks. Charlotte possesses excellent language skills.

Charlotte took 7 minutes 45 seconds to solve the first task, with hints, and then spent 10 minutes on the second task before giving up. At the end of the think aloud, she was shown the solution; hence the "4.provide" for the level of help given.

4.2 Charlotte – The First Shift Task

4.2.1 Excerpt 1

Summary

Charlotte began by reading the problem. She initially expressed a lack of familiarity with the nature of the task. However, it was quickly established that she thought she was required to provide code to move the boxes. (In retrospect, not as bizarre an interpretation as we first thought, given the GUIs that students are now accustomed to experiencing.) The interviewer clarified that the task was to write code to shift the values in the variables according to the arrows in the diagram. To establish that Charlotte did then understand the task, the interviewer asked Charlotte to choose some initial values for the variables and then determine the final values in the variables after her code had executed.

Data

Charlotte: So, may I ask is it similar to last week?

Interviewer: Yes, but instead of swapping two variables it's ...

Charlotte: … swapping 4. And I want them all to move to the left. So I'm moving the values not the variables. Ok good to know - makes more sense.

Analysis

In this excerpt, Charlotte made a connection between shifting and swapping values: where each requires "movement" of values between variables using assignment. Although she used the word "swap" which is a reciprocated exchange of values between two variables, she showed an understanding of the shifts required.

4.2.2 Excerpt 2

Summary

Charlotte made a first attempt to solve this task and although each assignment statement in itself was correct (apart from using a variable `t` instead of `temp`) the sequence of her assignment statements was not correct. She then traced the code using the values she had chosen for each of the variables: 2,4,6,8 and 10 for `a`,`b`,`c`,`d` and `t`. When she incorrectly concluded that the code worked as required, she was challenged, and then decided to re-read the question.

Data

<*Charlotte wrote the code below*>

```
a = b
b = c
c = d
d = t
t = a
```

Charlotte: So it almost worked… Oh no! I think it did work the way I wanted it to. So it says the `temp` becomes 2. Yeah I think that worked.

Interviewer: Where does the value 2 end up?

Charlotte: <*quoting the problem description*> "…with the left most value ending in the right most variable". Ah! It was cute while it lasted!

Analysis

Each assignment statement in Charlotte's solution was correct, but they were out of order. That is, she focused on parts of her solution while losing sight of the whole task, which is characteristic of reasoning at the preoperational stage. Neo-Piagetians refer to this phenomena as "spatial and temporal centration", or more colloquially, being unable to "see the forest for the trees".

4.2.3 Excerpt 3

Summary

Charlotte then realised that `a`'s value must first be temporarily stored so it will not be overwritten and lost. She was not convinced that her subsequent solution worked until she executed a trace of her code.

Data

Charlotte: Well we need `d` equal to…? Ok. So I'm trying to figure out where the temporary best comes in because what we really want at the end of the day is `t` to equal `a` from the beginning.. <*Charlotte then wrote the code below*>

```
t = a
a = b
b = c
c = d
d = t
```

So that works! I think...

Analysis

Charlotte realised the importance of sequence and figured out that `a`'s value must be saved *first*, so that that value can be assigned to `d` after `d`'s value has been reassigned. Charlotte made the leap from *individually correct* assignment statements to correctly sequenced lines of code in order to achieve the required effect. She was however heavily reliant on tracing the sequence with specific values to convince herself of the code's correctness, a manifestation of the preoperational stage of development.

4.3 Charlotte – The Second Shift Task

The second shift task was attempted by Charlotte in the same think aloud session where she completed the first.

4.3.1 Excerpt 4

Summary

Charlotte made a connection between this task and the previous task, but then had some doubt about their similarity when she read the supplied line of code. She established a set of initial values for each of the variables, and the expected final values for each.

Data

<*As Charlotte uttered what follows, she wrote the initial and expected values in the boxes of the supplied diagram.*>

```
Variables:     w  x  y  z
Initial:       2  4  6  8
Expected:      4  6  8  2
```

Charlotte: So it's the same as the first one. And then … here that `temp` equals `y`, now I'm really sceptical. Um, I don't think it actually is, so we'll find out. 2,4,6,8 <*values for variables* `w,x,y,` *and* `z` *respectively*> and we want to move everything to the left and the left most one ends up in the right most variable.

Analysis

Charlotte manifests preoperational behaviour by setting up specific variable values with which she intends to reason about her code. Another preoperational behaviour is her focus on the superficial aspect of the task, that is, the initial assignment to the `temp` variable.

4.3.2 Excerpt 5

Summary

Charlotte paused to question the reason for the supplied line of code, but after not being able to come up with an answer, started to implement a solution. With the first assignment of `y` to `temp`, she articulated its new value, 6. When she had finished writing the remaining assignment statements (shown below), she was not confident that her answer was correct, and expressed frustration. To the left of each of her lines of code, she wrote the value being assigned to the variable on the left of the assignment. When the values didn't match those expected, she realised her code must be incorrect.

Data

Charlotte: But you have to start with the `temp` as `y`. Why? Interesting question. … Fine. If you insist, `temp` is `y,` so `temp` becomes 6. … Where do I want it to go? Hmm. … Brain - wake up! … So … `x` to be `y` … Does that make sense? Ok for now it does. `w` to be `x` …`z` to be `w`. No we don't. Nnnnn, yes we do. … Aaargh!

```
6       temp = y;
6       x = y;
4       w = x;
4       z = w;
```

`z` becomes 4 which we do not want! Think I'm breaking the thing I realised before.

Analysis

Although incomplete, most of Charlotte's assignments were independently correct. However, the sequence of these assignments was not correct. She did not relate this second task to the approach she had successfully developed to solve the first task, but instead constructed assignment statements according to the diagram, in what appeared to be a random order. Charlotte was unable to make an accurate determination of the code's correctness until she traced it with specific values. Charlotte did not even trace her code accurately (in the third line she failed to take into account the updated value of `x`), and it was evident through utterances of contradiction ("No we don't. Nnnnn, yes we do.") and frustration ("Aaargh") that she was cognitively overloaded. Because Charlotte said "Think I'm breaking the thing I realised before", we hypothesise she had some hazy notion of the *invariant amid transformations* in this exercise, that is, that saving a variable's value to a temporary location makes it possible to overwrite that value in the original variable. This was the "thing" that her current solution was "breaking".

4.3.3 Excerpt 6

Summary

Charlotte made her final attempt before running out of time. On this occasion, she started reassigning from the far right of the line of variables in the diagram and again recorded the value being assigned at each statement.

Data

Charlotte: `z` equals `w`, which basically becomes 2. `y` … becomes `x` so that's 4. <*Expletive*> Sorry, `x` equals `y`. So if `x` equals `y`, that becomes 6.

```
6       temp = y;
2       z = w;
        ~~y = x;~~
6       x = y;
```

Um. Start over. `z` becomes `w`, that's good because that's 2. `x` becomes `y` which becomes 6 so that's good. … Too confused … We have to back off here a little bit.

```
6       temp = y;
2       z = w;
~~6       x = y;~~
```

So we want `w` to equal `x` … which basically becomes 4. I haven't removed `x`, the value of `x` yet. … I think that's where things were trying to click in because then `x` becomes `y` … and that becomes 6. `y` becomes `z` which becomes 8. … Well … wait - what's wrong with that? Why doesn't that work?

```
6       temp = y;
2       z = w;
4       w = x;
6       x = y;
8       y = z;
```

Ok and `z` because we said `z` is `w` up here, so why is that a problem? … because that's the problem! Grrrr! Ok, I think I have to go <*to another appointment*> …

Analysis

Charlotte's piecemeal approach to solving this task was not effective. She was focused on individual assignment statements and lost sight of the bigger picture (shifting all of the values without losing any of them). She was unable to work with the cyclic series of variables starting from an intermediate point. For all the reasons given with these excerpts, Charlotte is clearly at the preoperational stage of development.

4.4 Jim

It was the third week of semester when Jim performed the following think aloud on the first task. Furthermore, in an earlier semester, Jim had successfully completed a course that included about 6 weeks of programming in Python. In his think aloud sessions, Jim demonstrated adequate language and communication skills. Jim had completed one think aloud session with us prior to completing the first shift task which is described below.

4.5 Jim – The First Shift Task

4.5.1 Excerpt 7

Summary

Jim read the question text and then proceeded to select values for each of the five variables.

Data

Jim: So we can say that a is 1, b is 2, c is 3, d is 4. And following what this diagram says, we also have a fifth variable which we will call e, though in the diagram it's called temp. That will be the value of 5. Though it doesn't matter.

Analysis

The diagram stipulated that the temporary variable was called temp. It is odd that he chose to rename it e. When later queried, he said he was opting for consistency: the other variables had one letter identifiers, so he chose a one letter identifier for the temporary variable. Also odd was his subsequent use of capital letters for the other variable names, instead of the lower case used in the diagram. In any event, as will be shown below, his unusual choice of variable names had no effect on achieving the desired outcome on this first task.

Jim's reliance on specific values when reasoning about and writing code is characteristic of preoperational behaviour.

4.5.2 Excerpt 8

Summary

Jim articulated a logical sequence of assignment statements to complete the task, but was then not confident about his solution.

Data

Jim: So we want to move A first. So we want e to take the value of … A. Um. ... then we can say … that A can take the value of B. Um. C, uh B can take the value of C. C can take the value of D. And ... D can take the temp value. *<Jim had written the following>*

```
e = A
A = B
B = C
C = D
D = 5
```

...whoops. Going the wrong way around

Interviewer: Have you?

Jim: Oh no I haven't. So we want to go one more time around.

Interviewer: Do you?

Jim: To be … well, we want A to be stored over here *<indicating* D>

Interviewer: What's in D *at the moment?*

Jim Um, in D at the moment is a 5.

Interviewer: Why did you hard-code ... the number 5?

Jim: Um. I just assigned it a value. … I put 5 into D. I want A to go in there. So ... but A is now in e. Oops … that should be e. *<He then changed the code to the following.>*

```
e = A
A = B
B = C
C = D
D = ~~5~~ e
```

Interviewer: Are you finished?

Jim: Um, well I want A to be in D.

Interviewer: What's in D at the moment?

Jim: 5

Interviewer: Are you sure?

Jim: Yes

Analysis

Jim's first attempt is punctuated with hesitation, changes of mind, self-correction and finally an error he overlooks (the omission of the reassignment of the temporary variable's value). This behaviour is indicative of someone operating at the preoperational level. Jim rectifies his mistake, but only after prompting. Although his solution is correct, Jim did not reason about it accurately, as he thought that the original value of e (5) was assigned to D.

4.5.3 Excerpt 9

Summary

Jim was then asked to trace his code using the values he had already chosen. As he recounted each assignment statement's effect with specific values, it was only then that he articulated the temporary storage and subsequent reassignment of A which convinced him that the code was indeed correct.

Data

Jim: So, e equals A so e will equal 1. A equals B so A will equal 2. Um B equals C, so B will equal 3. Um C equals D so C will equal 4 and D equals e so D will equal … 1. Because e is equal to 1, that we'd gotten first at the top. … Ok. So it's not 5, it's 1. I see. So we have 1 in here <e> so that means we're going to have a 1 in here <D> now.

Analysis

Once Jim traced his code with specific values, he confirmed that his code was correct. Like most preoperational novices, Jim was not able to clearly reason in an abstract way about his code. He needed to trace the code with specific values in order to feel confident about its correctness.

4.6 Jim – The Second Shift Task

The second shift task was completed by Jim two weeks after he had done the first task. He took an enormous amount of time (more than 21 minutes) and several attempts to complete it. The following excerpts are only a small sample of Jim's articulations for this task, but are representative of the difficulties he had.

4.6.1 Excerpt 10

Summary

After reading the question, Jim immediately recognised this task as familiar. He expressed scepticism about the given initial assignment statement. He then allocated values to each of the variables, including temp (both in the diagram and in the given line of code) and then worked his way through the diagram, writing an assignment statement to match each shifting value. He then attempted to formulate the correct sequence of those assignment statements.

Data

Jim: temp is assigned y. … This seems slightly unnecessary …

Ok um. So temp's got the value of y ... So ... where are we... we've got ... let's say w equals 1, x equals 2, y equals 3, z equals 4. <*He wrote the following set of initial values*.>

```
w = 1
x = 2
y = 3
z = 4
```

So we want to move... we've got 1,2,3,4 … 3. No it's easy, we get rid of that y value because we've got two 3's. That means. So ... um we can just say … Ok ... so we want. ... start <*with*> the y. ... um …so we want ... so we want …1 …we want over here so we don't want z to equal, z equals 1 then the 4's going to disappear. If w equals x, the 2 is going to disappear. … If x, x equals y, the 3's still going to ... stay, so we can say... no the 2's going to disappear so we can say y equals z. ... So y equals z. <*He wrote the following single line*.>

```
y = z
```

So y equals z, so y will equal 4 now. So we've got 4 here … We can say… just wait. So still the left most variables ... why would we want to do that, why wouldn't we just say y equals … We need 3 so y equals… w. Going to move them all now. Um. What are we doing with this? I like to confuse myself a little bit. … And then we can have the 3 here, so it <z> is going to be ... um 4 <*recorded z as now having the value 3*>. … Yep. Ok. ... Um ... So we want x.... we want the z to equal w, we want w to equal z. … We want x to equal y, and we want y to equal z. <*He had written the following statements, separate from the previous single line of code*.>

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

So we've got y is equal to 4. So z is 3. So we want z to equal ... 1, want w to equal 2, we want x to equal 3, we want z … z to equal w. <*He revised the statements as follows*>

z = w

w = ~~z~~ x

x = y

~~z~~ y = z

So ... z is 4 so there we go <*wrote 4 under the y of y = z*>. That's a bit … that's a bit better. So y to equal z. It's annoying because it's so simple, but not. [laugh]. Just messes with your mind!

Analysis

Jim determined that the reassignment of y should be the first step, only after testing the effect of first reassigning to z, then to w and finally to x.

Jim has so far made hard work of this task by recording four separate sets of data. First, he allocated integer values to each of the variables by writing what appeared to be assignment statements. Second, he wrote the beginning of an ordered sequence using those assignment statements. Third, he wrote an assignment statement for each "shift", starting from the right hand side of the diagram. In addition, Jim kept current trace values recorded under several variable names in the code.

Jim is dependent on reasoning with specific values in variables. With his trace notation interspersed in the code it was very difficult for him to follow on paper what he had written, let alone keep track of what he had left to do. When speaking, he repeatedly intermingled variables and values when referring to what needed to be assigned where. He made several contradictions by saying one thing and writing another. He showed some confusion about assignment direction, repeatedly changed his mind and made tracing errors throughout.

Jim was clearly cognitively overloaded, unable to manipulate the abstraction of the diagram in such a way that it represented a solution that started with the reassignment of y, and unable to design an effective trace of his code. These are all indicative behaviours of someone at the preoperational stage of development. Indeed, his haphazard approach to tracing is a characteristic of the sensorimotor stage. Although he did articulate an abstraction beyond the code itself, the need to "get rid of that y value because we've got two 3's", he did not continue to apply that principal to the remaining variables, as he had successfully done in the first task. Not applying an abstraction consistently and completely is characteristic of a preoperational novice.

4.6.2 Excerpt 11

Summary

At this stage, Jim had established expected final values for each of the variables, using the initial values he had chosen. After having painfully determined by trial and error what the first assignment should be, he struggled to establish a workable sequence of the remaining assignment statements.

Data

Jim: We want … x to equal the…3 so it currently holds the third value in temp. So we can say x equals temp. … So x has now got the third value. … temp is still empty so we can say... so we've got x and y sorted. Just need w. What do we want `w` to equal? Whoops! <*he exclaims while crossing out the third row below*>. That shouldn't be there because it gets rid of my 2 value.

```
temp = y
y = z
```
~~`x = temp`~~

So we need to store ... `w` in the `temp`. … `temp`'s got the value of `w` so now we can ... that `w` value. … So that `w` value we want to equal 2... so we want `w` to take the value of `x`. So the `w` value's been wiped ... being stored in `temp`, so the `w` value is given the value of 2 that should still be 2

```
temp = y
y = z
temp = w
w = x
```

[sigh] … I think I just lost my ... lost my 3 then. Yeah, I've lost my 3 [sigh] Ah, it's frustrating!

Analysis

Jim correctly dealt with the reassignment to `y` after which he focused attention on the start of the series rather than continuing from that intermediate point. He struggled to implement the logic that he used successfully two weeks earlier on the first shift task.

In the first line of this excerpt, Jim refers to the "third" value, so we suspect that he saw the ordering of the variables in the diagram as significant. After dealing with the reassignment of `y` as required, he found it necessary to continue at the start of the diagram. This may explain his comment in Excerpt 10 that he found the forced assignment of `y` to `temp` as "slightly unnecessary". As a preoperational novice, he was unable to effectively apply the invariant of saving a variable’s value for subsequent reassignment. He had completed the first task successfully, but was unable to mentally manipulate the new diagram in such a way that it replicated the first, that is with `y` at the beginning of the reassignment sequence, rather than in the middle.

4.6.3 Excerpt 12

Summary

Jim made several other failed attempts at this task, experimenting with different values stored in `temp`, but each time articulating a trace of the real values he had chosen. At a point where he was clearly frustrated, the interviewer suggested that he stop concurrently tracking the variables' values while developing the code, thus eliminating what seemed to be a distraction.

Data

Jim: This is starting to frustrate me a little bit. [laugh] I'm not going to lie. Seems so much more um... I don't know ... difficult. When you're not doing it on the computer. What I'm saying is that ... like... if you don't have the numbers there... you can ... I think numbers helps so you don't accidentally clear them.

Interviewer: *when you did this last week you ... stored one of the values away to start with. Why?*

Jim: ...Um, well I don't remember [laugh]

Interviewer: *You don't remember why?*

Jim: Um, just so it didn't get cleared. Ah, I see! …Same as last week. I see ... But I'm just … See what I'm trying to do, I'm trying to rearrange the numbers because I'm saying if its 1,2,3,4 .... and we've got the 3 in here <i.e. in `temp`>...

Interviewer: *So WHY do you have a 3 in there?*

Jim: Because the `y` is equal to `temp`. So, if I call <`y`> 3, then <`temp`>'s going to be 3

Interviewer: *So then what's your first step?*

Jim: So the first step ... I can move the `z` to <`y`> ... And then I can move <`x`> to <`w`>... sorry, no I can move <`w`> to the `temp`. …

Interviewer: *... when you did this last week, how many temp variables did you use?*

Jim: One

Interviewer: *So why should this be any different?*

Jim: I don't know. … These <*tasks*> ... they're like a lot easier than the programming that I'm doing, but they're a lot harder at the same time. It's just different - it's weird. [laugh] It's not nice. It confuses me.

Analysis

Jim continued to have trouble with this task which forced him to start from an intermediate point, that is, the required initialisation of `temp`. In the first task he appeared to have demonstrated an understanding of the process required to shift the values in four variables as well as the programming skills to implement it. However, without prompting by the interviewer, he had an enormous amount of difficulty transferring that (possible) understanding of a very similar task. His level of ability in terms of abstract reasoning was clearly preoperational.

4.7 Steve

Steve’s think aloud sessions were indicative of concrete operational reasoning. Steve was in his first semester of learning to program. He completed his first think aloud session in week 3 of semester.

4.7.1 Excerpt 13

Summary

After needing initial clarification of the diagram, Steve completed the first task in a matter of seconds.

Data

Steve: So a will become d and d will become a

Interviewer: *Ah, the value in a will go into d - like this diagram shows, the value of a eventually goes to d.*

Steve: and d eventually goes to a.

Interviewer: *...c goes into b, b goes into a...*

Steve:	Ah, so shuffle it along.
Interviewer:	*Yeah. Move everything up to the left*
Steve:	Ok so. ... temp equals a. a equals b. b equals c. c equals d. d equals temp.

Analysis

Steve's initial interpretation of the first task was that the values in variables a and d were to be swapped, with the top arrows in the diagram indicating the passing of d's value through c and b, and finally ending up in a. His understanding was quickly corrected, confirmed by his articulation of the task as a 'shuffle' and then immediately writing a correct solution.

4.7.2 Excerpt 14

Summary

Steve then attempted the second task, and completed it without hesitation:

Data

Steve:	Ok. .... temp equals y so we've stored the y value. So then we can replace it with the z value. Yes. y equals z. Then you replace the z value with w. w value with x ... And then. Ah yeah, then x value with the temp

Analysis

Steve had clearly identified the invariant: "temp equals y so we've stored the y value". He applied the same process of storing a value before overwriting the variable with what was to replace it, for the remainder of the variables. With concrete operational skills, Steve had no problem applying the skills he used in the first task to the slightly different second task.

5 Discussion

During these think aloud sessions, we noticed variation in the way that some students articulated assignment statements. For example, with respect to the following assignment statement:

```
a = b
```

some students articulated the statement from *left to right,* thus:

"a *is assigned* the value of b"

others read from *right to left, that is*:

"the value of b *is assigned to* a"

while others articulated assignments both ways: sometimes *left to right* and sometimes *right to left.* We conjecture that such variation in articulation is indicative of novices at a neo-Piagetian stage lower than concrete operational.

During the think aloud sessions, it also became apparent that some students struggled to process the diagrammatic depiction of the problem. One possible problem was the direction of value "shifts", as the majority of the values passed between variables *right to left,* but the value originally in the leftmost variable moved *left to right.* Some of the students even expressed confusion over the meaning of the arrows. Apparently it was not immediately clear (as it was to us, and probably to any experienced programmer) that the arrows indicate the direction of movement of the values.

The think aloud students who struggled with the second shift problem tended to look at a small part of the diagram and implement it. Next they would return to the diagram and find another piece to implement, without much thought to the consequences of sequential execution. They had not developed an overall design for their solution, but instead focussed on the functionality for each independent piece of the problem, in the hope that they would somehow all fit together in the end. Being distracted from the most salient aspects of the problem by individual elements is characteristic of preoperational reasoning.

Even some students who completed the second task quickly expressed some awkwardness about it. Lance said "*That felt weird. I didn't really like having to start there. Don't know why*." Becki said that the second task was "*very sneaky*" and it had ruined her plan to start from the end as she had in the first task. She also said that it would not have made a fundamental difference had the diagram depicted the variables in a circle as the variable names were ordered and she tended to work on the variables in lexicographic order. However, despite some initial and brief confusion, these students were able to complete the task. Students like Lance, Becki and Steve thus manifested concrete operational skills.

6 Conclusion

In this paper, we have presented data from a think aloud study which demonstrates that some novice programmers manifest behaviours characteristic of the preoperational stage in neo-Piagetian theory. One such behaviour is that they tend to focus on parts of a programming task and lose sight of the task as a whole. Students who struggled with the second "shift" task tended to examine a portion of the diagram and implement it, then return to the diagram and find another portion to implement, and so on, without considering the overall sequence of execution.

Another characteristic of these preoperational novices is that they are prone to focus on superficial aspects of a specific task that are not salient to solving a general class of tasks. In neo-Piagetian terms, preoperational novices do not focus upon aspects of tasks that are "*invariant amid transformations*" (Flavell 1977). In the "shift" tasks, the invariant is the idea of duplicating a variable, so that the value in the original variable might then be overwritten, while the superficial aspect of the task is the initial assignment to the temp variable.

These two characteristics lead preoperational novices to adopt an approach that might be called *programming by permutation.* On very small tasks, that approach may indeed lead the novice to a correct solution, especially if they are completing that small task on a computer and thus receive feedback by running their code. However, novices who adopt that approach do not learn abstractions that they can then transfer to a very similar task.

The two "shift" tasks we gave our students are very simple programming tasks, the solution for which is near-identical in most imperative languages. The problems experienced by some of our novices are therefore not caused by the particular programming language in which they write.

Piaget (1971b) described reasoning at the preoperational stage as that "*... which consists simply in retracing ... events just as they were perceived, instead of imagining an alteration ...* ". It is only at the concrete stage of development that novices develop the ability to work with cyclic series, to reason about abstract quantities that are conserved, and transfer a general approach to a slightly different task.

When students demonstrate difficulties with programming, it may not be a reflection of their innate ability to learn programming, but rather an indication of their current state of cognitive development. Struggling students may not have yet developed the mental schemas necessary to perform at the concrete operational level of reasoning required by certain programming tasks.

On the basis of our qualitative work, we cannot draw firm conclusions about the commonality of preoperational reasoning. However, given that four of our eleven think aloud volunteers manifested this difficulty, it is possible that preoperational reasoning may be common. Further quantitative work is warranted. If future studies confirm that this is a widespread issue among novice programmers, then it suggests that our teaching practices should change. The change would place the focus on identifying the current neo-Piagetian stage of a novice, and provide tuition appropriate to moving that novice to the next stage. Current pedagogical practice places little emphasis on the sensorimotor stage and completely ignores the preoperational stage. That is, current pedagogical practice assumes that the basic programming constructs are learnt easily, and then students immediately begin to reason about programs at the concrete operational stage.

7 Acknowledgments

The authors thank the students who volunteered for this study. 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

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 (ACE 2012)*, Melbourne, Australia. **123:**77-86, ACS.

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.

Flavell, J. H. (1977): *Cognitive Development*. Englewood Cliffs, NJ: Prentice Hall.

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

Perkins, D. N. and Martin, F. (1986): Fragile Knowledge and Neglected Strategies in Novice Programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers*. Norwood, New Jersey: Ablex Publishing Corporation.

Piaget, J. (1971a): Order of Succession Inherent in Cyclic Movements. *Chapter 2 of The Child's Conception of Movement and Speed* 37-60. New York: Ballantine Books.

Piaget, J. (1971b): Problem of Alternative Directions of Travel. *Chapter 1 of The Child's Conception of Movement and Speed* 3-36. New York: Ballantine Books.

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 (6th ed)* Vol. 2: Cognition, Perception and Language, 464-510. Hoboken, NJ: Wiley.

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.