ABSTRACT
In this paper, we look at the concept of reversibility, that is, negating opposites, counterbalances, and actions that can be reversed. Piaget identified reversibility as an indicator of the ability to reason at a concrete operational level. We investigate to what degree novice programmers manifest the ability to work with this concept of reversibility by providing them with a small piece of code and then asking them to write code that undoes the effect of that code. On testing entire cohorts of students in their first year of learning to program, we found an overwhelming majority of them could not cope with such a concept. We then conducted think aloud studies of novices where we observed them working on this task and analyzed their contrasting abilities to deal with it. The results of this study demonstrate the need for better understanding our students’ reasoning abilities, and a teaching model aimed at that level of reality.

Categories and Subject Descriptors
K.3 [Computers & Education]: Computer & Information Science Education – Computer Science Education

General Terms
Measurement, Performance, Experimentation, Human Factors

Keywords
Neo-Piagetian theory, programming, novice programmers, think aloud, reversibility.

1. INTRODUCTION
Piaget used an experiment using a bottle of coloured water which he rotated at various angles under cover so only the outline of the bottle could be seen. He asked children to draw the changing water level as he rotated the bottle back and forth, hence testing their understanding of the effect of reversing a process or effect.

We can think of reversibility in a programming context as cancelling an effect by reversing the steps in order to revert to an original state. We followed Listers’s [4] suggestion of providing code that rotates the values in an array in one direction with the task being to write code that rotates the values in the opposite direction, as shown in Figure 1. As the programming languages varied between semester offerings of our units, some students completed these exercises in C#, rather than Python (see C# solution in Figure 2). At our institution, we tested students’ ability to reason about reversibility in this way, across several cohorts. Our students were introduced to lists and indexing in week 2 of semester, and start working with loops by week 5.

The purpose of the block of code below is to take a list of numbers containing five integers and to move all elements of the list one place to the right, with the rightmost element moving around to the leftmost position.

```python
temp = numbers[len(numbers) - 1]
for index in range(len(numbers) - 1, 0, -1):
    numbers[index] = numbers[index - 1]
numbers[0] = temp
```

For example if numbers initially has the value [1, 2, 3, 4, 5] then after the code has executed, it would contain [5, 1, 2, 3, 4]. If we were to show the effect of moving all the elements of a list in this way in a diagram, it would look something like this:

```
*numbers*

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>e</td>
<td>m</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>
```

Note that `range(len(numbers)-1,0,-1)` produces a list containing numbers from one less than the length of the list down to one. For example for numbers defined above, the following list would be produced [4, 3, 2, 1].

Write Python code that does the opposite of the above code. That is, write code to move all elements of the list numbers one place to the left, with the leftmost element being moved to the rightmost position.

Sample Python Solution:

```python
temp = numbers[0]
for index in range(len(numbers) -1):
    numbers[index] = numbers[index + 1]
numbers[len(numbers) - 1] = temp
```

Sample C# Solution:

```c#
int temp = values[0];
for (int i = 0; i < values.Length -1; i++)
    values[i] = values[i + 1];
values[values.Length -1] = temp;
```

We tested our students with this task at the end of each of our first and second programming units and we were astounded by the consistently poor results (see Table 1 below). In fact, after an additional semester of learning programming, students were no more likely to be able to complete this task. We had simply asked them to modify some relatively familiar code to change its functionality—and they could not. The best results achieved by any of our cohorts was 68% incorrect answers. That is, less than a third could write that code.

Table 1: Cohort Testing of Reversibility Task

<table>
<thead>
<tr>
<th>Weeks of programming instruction completed</th>
<th>Number of students</th>
<th>Percentage wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>71</td>
<td>92%</td>
</tr>
<tr>
<td>12</td>
<td>322</td>
<td>68%</td>
</tr>
<tr>
<td>24</td>
<td>68</td>
<td>79%</td>
</tr>
<tr>
<td>24</td>
<td>60</td>
<td>85%</td>
</tr>
<tr>
<td>20</td>
<td>82</td>
<td>78%</td>
</tr>
</tbody>
</table>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires specific permission and/or a fee. Request permissions from permissions@acm.org.

Copyright © 2014 ACM 978-1-4503-2833-3/14/06...$15.00.
http://dx.doi.org/10.1145/2591708.2591712
While grading these tasks we ignored minor syntax issues like missing brackets or colons, misplaced indenting, inconsistent variable names etc. However, the 5 criteria which we deemed necessary in order to mark as correct were:

a) line 1: correct value stored in temporary variable
b) line 2: correct for loop structure
c) lines 2 & 3: use of valid range of subscripts
d) line 3: correct rotation direction
e) line 4: correct relocation of temporarily stored value

To understand why so many students could not correctly complete a task which we thought was relatively simple, we called for volunteer students to take part in studies where we gave them this reversibility task to complete while thinking aloud. We then interpreted our data using the neo-Piagetian theoretical framework.

2. NEO-PIAGETIAN STAGES

Neo-Piagetian theory was developed as a response to problems identified in classical Piagetian theory. Both classical and neo-Piagetian theories describe cognitive development in terms of sequential, cumulative stages. However, unlike classical Piagetian theory, Neo-Piagetian theory is not age-related and applies equally to adults. Furthermore neo-Piagetian theory describes cognitive development as a person acquires knowledge in a specific domain. The cognitive structures become more complex through exposure in the domain, which explains why someone is capable of reasoning at different levels of abstraction in two different domains (e.g. Math and literature). Lister [4] proposed that these stages are evident in the novice programmer.

At the least mature stage, sensorimotor, there are plenty of misconceptions about programming which are not necessarily applied consistently. The sensorimotor programmer at best possesses fragile domain knowledge as disjointed snippets which he finds difficult to piece together in any satisfactory manner. At this stage the novice is focused on the mechanics and syntax of the language, and tasks like tracing code require considerable cognitive effort. He manipulates code by trial and error.

At the next more mature stage, preoperational, a novice is able to more reliably trace code, but any attempt to reason about it is only intuitive. Misconceptions that remain are at least applied consistently. A preoperational novice writes and traces code with heavy reliance on and reference to specific values. They are incapable at this stage of seeing a relationship between different parts of code, or of seeing how the parts fit into the whole. A novice at this stage does not really understand actions so their thinking is dominated by static images rather than mental representations of change [2]. In terms of the reversibility problem in Figure 1 and Figure 2, a preoperational novice does see an immediate relationship between the code they were given and the code they must write. They can usually write their answer without hesitation.

We do not however suggest that there is necessarily a quantum leap from one stage to the next, or that we can classify novices as being at one particular stage of development. Rather than a one-way staircase model of development, we adopt the overlapping wave metaphor of Siegler [7], where as a person acquires knowledge and skills in a new cognitive domain, they exhibit a changing mix of reasoning strategies from different stages. Firstly they reason predominantly at a sensorimotor level but as they build schemas and develop skills in the domain they reason less at that level and more at the next more mature level, and so on. In this way, multiple ways of reasoning can coexist.

3. THINK ALOUD STUDIES

Using this neo-Piagetian framework we gathered think aloud data from students as they completed the exercise in either Figure 1 or Figure 2 to help us understand why many students could not reverse the effects of a piece of code.

3.1 Think Aloud Results

We discovered that our think aloud students exhibited behaviours representing a broad range of neo-Piagetian stages, but that a student’s cognitive maturity was not necessarily commensurate with the amount of progress they had made through their programming course. We now focus on a small yet representative selection of these students, summarized in Table 2. We have chosen three students who were clearly demonstrating characteristics of one of the first three neo-Piagetian stages (sensorimotor, preoperational or concrete operational), and one that seemed to be on the cusp of progressing from preoperational to concrete operational. For each student, Table 2 shows the aggregate number of weeks they have completed of programming instruction.

Table 2: Think Aloud Performance on Reversibility Task

<table>
<thead>
<tr>
<th>Alias</th>
<th>Weeks of instruction</th>
<th>Time taken (m:s)</th>
<th>Neo-Piagetian Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>45</td>
<td>3:51</td>
<td>Concrete</td>
</tr>
<tr>
<td>S2</td>
<td>11</td>
<td>14:58</td>
<td>Preop-Concrete</td>
</tr>
<tr>
<td>S3</td>
<td>9</td>
<td>13:20</td>
<td>Preoperational</td>
</tr>
<tr>
<td>S4</td>
<td>14</td>
<td>11:15</td>
<td>Sensorimotor</td>
</tr>
</tbody>
</table>

3.2 Analysis of Think Alouds

In this section we dissect the think aloud transcripts of those four sessions, which illustrate why students struggle with this reversibility problem. We document these excerpts in a format similar to that used by Lewis [3] and more recently by Teague and Lister [8] who separated the interview data from its analysis in order to help the reader better follow the participant’s progress during the session.

Pauses in speech are marked “…”, as are placeholders for deleted utterances which we deemed unnecessary for inclusion as those utterances add nothing to the context of the think aloud session.

We start our think aloud sessions with one which exemplifies a concrete operational student, or in other words, a student capable of operating at a fairly mature level.
3.2.1 S1 - Concrete – Week 45: C#

3.2.1.1 Summary
S1 had little difficulty completing this task. He read the question and jumped straight into writing his solution. He completed the exercise in less than 4 minutes.

3.2.1.2 Data
S1 started writing his solution by copying the given code. Before completing the first line he realized he had copied too much (i.e. the index of the rightmost element of the array) and immediately self-corrected. As he changed the subscript to 0, as shown in Figure 3, he said:

S1: *I want... to get... 0 because we want the first thing to be at the end*

He then completed the two lines of the *for* loop, correctly, without hesitation, as shown in Figure 4. S1 gave an abstract explanation of the reassignment of values which would affect movement in the correct direction:

S1: *... and now I simply grab what is one ahead of it and move it back 1 so... values... at i is assigned... values at... i plus 1 which is going to the next one over....*

3.2.1.3 Analysis
S1 illustrates what most computer educators expect from their students. The abstract manner in which S1 described code is consistent with someone operating at the concrete operational level. He does not rely on tracing with specific values in variables or specific indexes in order to understand code, nor does he refer to specific values when writing code.

3.2.2 S2 – Preop-Concrete – Week 11: Python

3.2.2.1 Summary
S2 read the question text very slowly, then partially copied the given code and started annotating it. He drew arrows to indicate the direction of movement of the values and added specific values with which to trace that movement. After taking more than four and a half minutes to do this, he started writing his solution. He used diagrams and specific values to partially trace his code and, except for an off-by-one error, his final solution was correct. He articulated that he had found the exercise troublesome. He had taken nearly 15 minutes.

3.2.2.2 Data
On reading the first line of the given code S2 immediately articulated an abstraction:

S2: *Temp equals list of numbers... at a place one before the end*

This abstraction was off-by-one, yet a note he wrote soon after (see Figure 6) was accurate.

3.2.2.3 Analysis
S2 started writing his code for the solution, quickly self-correcting any errors. As shown in Figure 8, he left blank the range of indexes to be iterated over, as well as the index of the value to be assigned inside the loop. He drew circles around both these blank parts after saying:

S2: *... figure that out later*

He completed the storage of the first value to temp and its subsequent reassignment into the last element without hesitation.
S2: so if it’s 4 ... so in place 4 we want 3 ... that’s what the original does

He then drew (see Figure 10) what he wanted his solution to do.

![Figure 10: S2 - Annotation of proposed code](image)

S2: I want ... if it’s 4 ... I replace 3 ... of index ... to have ... place 4’s value ... is that right [pause 12 seconds] 3 ... that’s moving it left ... so that’s what I want.

S2 was able to then complete the code which was correct, apart from two off-by-one errors.

S2: I had a lot of trouble with that one

3.2.2.3 Analysis
S2 completed the temporary storage of the end value and its reassignment at the other end before completing the loop. Once he had determined the indexes involved in the reassignment, he was able to complete the loop header, but he was only able to resolve this uncertainty with the use of diagrams and specific index values.

Unlike S1 who used no specific values while reasoning about the given code or writing his solution, S2’s reliance on specific values to formulate code is indicative of preoperational thinking. However, those operating at a purely preoperational level also find it necessary to trace code (or an algorithm) in its entirety, whereas S2 was manifesting nascent concrete operational skills by needing only a partial trace of the movement of values for verification.

3.2.3 S3 – Preoperational – Week 9: Python

3.2.3.1 Summary
S3 took nearly 13 and a half minutes to complete this task.

She wrote an initial solution which contained a correct for loop (except for off-by-one error) but which also contained the first and last line which were copied from the given code. She then corrected the first and last lines.

3.2.3.2 Data
S3 did not read the question out loud. She simply said enigmatically:

S3: list of numbers

After a long pause (68 seconds) she wrote her first line of code and then after another long pause (57 seconds) annotated her code by writing numbers above that line (see Figure 11).

![Figure 11: S3 - Temporary storage [incorrect]](image)

After yet another pause (34 seconds) she started writing the for loop, as shown in Figure 12. She hesitated when deciding the range of the iteration but finally settled on it iterating from index 1 to the end of the array.

![Figure 12: S3 - Loop structure [correct]](image)

S3 then wrote the body of the for loop as shown in Figure 13. However this line of code and the for loop header in combination produce an off-by-one error.

![Figure 13: S3 - Loop body [correct, in isolation, except for minor syntax issue (missing bracket)]](image)

Then she wrote an incorrect version of the final line of her code as shown in Figure 14.

![Figure 14: S3 - Relocation of temp value [incorrect]](image)

After a long pause of over 5 minutes, she was prompted by the interviewer to talk about what she was thinking.

S3: Um ... so I’m just trying to work out the ... temp variable, make sure that I’ve got it picking up the right number ... equals ... numbers um ... [sigh] so just going to change that to ... 0 ... have to pick up the 5

As she completed this utterance and changed her code, she also added a comment to her first line (see Figure 15).

![Figure 15: S3 – Revised temporary storage [correct, but the comment refers to a specific set of data values]](image)

S3 re-read the loop structure she had written, and reasoned about its correctness:

S3: so skipping the first number in the list ... to um the end of the list ... the index 0 here um ... would be 1 equals numbers at the index plus 1 so in this case ... the first time around ... we’re going to get ... 1 ... and then the second time around it will be 1 so we’ll get 2 ... and 3 and then 4

She wrote “1,2,3,4” in the body of the loop as a comment as shown in Figure 16.

![Figure 16: S3 - Commented loop body [correct, except the comment refers to specific data values]](image)

S3 then altered her last line of code as shown in Figure 17.

![Figure 17: S3 - Revised relocation of temporary value [correct, except for off-by-one error]](image)
S3 then reviewed her first line of code and added to the comment (see Figure 18).

**S3:** ok so 1 2 3 4 ... equals ... temp is 5

![Figure 18: S3 - Revised comment](image)

**S3:** I'm saying its 5 ... so if... index is 1 it's going to be ... index is 0 ... going to be 1 ... 1 2 3 4 ... gets around to 5

### 3.2.3.3 Analysis
In her initial solution, S3’s first and last lines of code were incorrect in that they moved the rightmost element to the leftmost position, while her loop was moving values from left to right. This is a classic indication of preoperational thinking, where two pieces of the code are written which are conceptually incompatible.

S3 focused on the salient aspect of the task. That is, she focused on the reversal of the direction of rotation. This is in contrast to S2’s approach which saw him initially focus on the temporary storage and relocation of the first value. In both cases however, the novices focused on detail which obscured the big picture. This behaviour is characteristic of preoperational reasoning.

While S3 did work with specific values including annotations to her code, her focus was frequently upon the collective direction of movement of those values. Thus she was reasoning at a slightly higher level than the movement of a single specific value at a time, but at a much lower level than S1 who traced abstractly.

### 3.2.4 S4 - Sensorimotor – Week 14: Python

#### 3.2.4.1 Summary
S4 had previously attempted this exercise two weeks prior to this session. At that earlier session, his solution had amounted to little more than a near-copy of the given code, except for removing the minus operator from the first line, which he believed to be responsible for the direction of rotation.

In the session detailed here, he struggled to even trace the code given in the question, which explains why he had previously not been able to write the reversal. He talked about the code in terms of individual symbols, with the focus clearly on syntax. S4 was unable to offer much explanation about the purpose of the code. He simply reiterated the question text.

#### 3.2.4.2 Data
S4 read the first line of given code, but he gave an incorrect articulation of the code’s purpose:

**S4:** numbers in brackets length of numbers minus 1 is assigned to temp ... so what’s really being assigned is ... ah ... I guess what was said up the top ... well moving 5 one [place] to the ... right most number to the left most position

The interviewer gave S4 a trace table with three columns headed `temp`, `index` and `numbers`. The initial value of `numbers` was recorded for him by the interviewer as “[1, 2, 3, 4, 5]”. No other data values were provided. When asked to trace the given code, he said:

**S4:** So numbers, length of numbers minus 1, so I think, ... well one off the length of numbers could mean there’s only four numbers ... soooo ... that would be 1,2,3,4.

Figure 19 shows what he then wrote in the trace table. Note that he has placed a list of values in the variable `temp` which indicates a misconception about the first line of code.

![Figure 19: S4 - trace table line 1 [incorrect]](image)

He continued with the trace as shown in Figure 20. Note that S4 incorrectly updated the temp variable (which is not even mentioned in the for loop) instead of the numbers list.

![Figure 20: S4 - trace table complete](image)

S4’s think aloud session lasted for more than 11 minutes and the entire session consisted of attempting to trace the given code. He did not attempt to write a solution.

#### 3.2.4.3 Analysis
S4 read the code symbol by symbol: “numbers in brackets length of numbers minus 1”. He did not provide any interpretation of the code or embellish on its purpose at all. Contrast that with S2 who articulated the same line as “numbers ... at a place one before the end”). S4 was not reasoning about the code at all, let alone as a whole. He was, as a child would when reading words, not connecting them into sentences. This behaviour is representative of someone at the sensorimotor stage of development.

S4 manifested a number of misconceptions, such as assigning a list to the variable `temp` and updating `temp` within the loop.

### 4. CONCLUSION
Listening to our think aloud students articulate about code provides evidence about their ability to reason abstractly. S4 used only specific values when he referred to elements and indexes in the array. S3 used specific values for both indexes and list elements as she developed her code. S2 referred to specific index numbers when he discussed the movement of values, but had the ability to talk purely in abstract terms (“Temp equals list of numbers ... at a place one before the end”). S1 mostly reasoned in the abstract (“grab what is one ahead of it and move it back”).

There is stark contrast between the performance of S4 and any of the other think aloud participants. Readers might conclude that this student showed little innate ability for programming, and that he would have most likely failed to progress much further in Computer Science studies. Conversely, S1 possessed exactly those abstract reasoning skills that would allow him to master programming at a high level. In fact, S4 is the same student as S1! The think aloud sessions with this student occurred one year apart.

Novice programmers show increased competencies with programming tasks as their ability to reason abstractly matures. But why is it that some of our students are working at such an immature level of abstract reasoning for so long? We believe that many students remain at the sensorimotor and preoperational levels because all the instruction they receive is at the concrete operational level. In other words, as Shayer [6] put it:
Think hard about the ways that your target students can process reality, and give them learning experiences which are accessible to them.

Students at the earlier sensorimotor and pre-operational stages require exposure to programming tasks at those levels before they can be expected to reason abstractly about programming. Vygotsky’s [9] theory of “zone of proximal development” might suggest pushing the boundaries of students’ current ability to reason, but to expect students to work with concepts as yet far out of their cognitive reach may also retard their progress by depleting both their confidence and motivation.

5. PEDAGOGICAL IMPLICATIONS

S1 represents the students to which we currently teach: students who are operating comfortably at the concrete operational level. It is at this stage when they can reason abstractly and work with concepts like reversibility. They are capable of dealing with non-trivial programming tasks.

How should we teach students at the preoperational level like S2 and S3? These two are probably more representative of our students. They are reliant on specific values to reason about code and require exposure to small tasks which are constituted from a minimal number of parts.

Other studies have found a relationship between reading and explaining skills and the ability to write code ([10], [11], [12]). Tracing code requires students to draw on the programming knowledge they accumulated at the sensorimotor level and explaining the purpose of code forces them to conceive a ‘big picture’, rather than remain focused on individual lines of code. Therefore, the progression of preoperational students into more mature concrete operational reasoning could be scaffolded with a sequence of these types of tasks that culminate in something like the reversibility question in Figure 1 and Figure 2.

... knowledge is not at all the same thing as making a figurative copy of reality for oneself, but that it invariably consists in operative processes leading to a transformation of reality, either in actions or in thought ... [5].

How then should we teach students at the sensorimotor level, like S4? These students have little hope of progressing until the misconceptions have been addressed. Many of these misconceptions can be identified and eliminated by a vigilant tutor or TA, but if left untended can fester into a quagmire of confusion and frustration. Sensorimotor novices should be exposed to only very small tasks with single elements on which to concentrate. We believe that at the sensorimotor level in particular, greater emphasis should be placed on reading and tracing code until misconceptions have been addressed. It is only then that students can be expected to start constructing their own snippets of code.

It is yet unknown how long it takes to progress from one neo-Piagetian stage to the next, but it is clear from this research that it is different for every student. What we do know is that the stages are sequential and cumulative. Therefore, we must allow students to master programming skills using the type of reasoning indicative of the earlier stages of development before exposing them to significantly more abstract concepts. The pedagogical challenge now is in the identification of students’ operational levels so we can facilitate a match of the appropriate levels with the learning material.

The reader may well be inclined to discount our results as unique to our institution. However, we challenge such academics to examine their own students’ ability to solve the same problem and report their findings.

6. ACKNOWLEDGMENTS

We would very much like to thank the students who took part in this study and the academics who allowed us to test their cohorts of students. In particular our gratitude goes to the think aloud participants who attended each week, especially those who participated for multiple teaching periods.

7. REFERENCES