Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer

Raymond Lister
Faculty of Engineering and Information Technology
University of Technology, Sydney
Sydney,
NSW, Australia

Raymond.Lister@uts.edu.au

Abstract
This paper brings together a number of empirical research results on novice programmers, using a neo-Piagetian theoretical framework. While there already exists literature connecting programming with classical Piagetian theory, in this paper we apply neo-Piagetian theory. Using that neo-Piagetian perspective, we offer an explanation as to why attempts to predict ability for programming via classical Piagetian tests have yielded mixed results. We offer a neo-Piagetian explanation of some of the previously puzzling observations about novice programmers, such as why many of them make little use of diagrams, and why they often manifest a non-systematic approach to writing programs. We also develop the relatively unexplored relationship between concrete operational reasoning and programming, by exploring concepts such as conservation and reversibility.

Keywords: Neo-Piagetian, Novice Programmer.

1 Introduction
After 30 years of teaching computer science and software engineering, at elite institutions like London's Imperial College, Kramer (2007) articulated a common lament of computing academics everywhere:

> *Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot? Is it possible to improve these skills through education and training?* (p. 37).

Kramer went on to claim that *critical to these questions is the notion of abstraction*. In his Turing Award Lecture, Dijkstra (1992) offered an explanation as to why the ability to abstract is important for programmers:

> *... the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called 'abstraction'; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.* (p. 864).

Both Kramer and Dijkstra were expressing a widespread belief in the computing community. For example, the ACM's Computing Curricula 1991 (Turner, 1991) includes the statement that "*... the process of abstraction will normally be prominent in all undergraduate curricula"*.

In his paper, Kramer drew upon the work of Jean Piaget, who developed a theory about the different levels of abstract reasoning exhibited by people, especially as they mature from child to adult. To perform the abstractions that Kramer and Dijkstra describe, a person must achieve the most abstract of Piaget's levels, which is called formal operational reasoning, but Kramer cites well known research claiming that only 30−35% of adolescents have achieved that stage, and some adults never achieve that stage. Kramer went on to advocate that students who apply to study computing at university should be tested for abstraction ability, and those students who test poorly should be refused admission to computing.

In this paper, we further explore the link between Piagetian theory and programming. Unlike most of the earlier work that has also explored that link, our perspective is not based upon classical Piagetian theory (i.e. as articulated by Piaget), but instead is based upon neo-Piagetian theory. The distinction between classical and neo-Piagetian theory is described in the next subsection of the introduction.

In section 2, we describe formal operational reasoning, the most abstract type of Piagetian reasoning, and the type most commonly discussed in the literature that connects programming with classical Piagetian theory. We review and re-evaluate that literature from a neo-Piagetian perspective.

In section 3, we describe preoperational reasoning, which is the least abstract type of Piagetian reasoning discussed in this paper. We reinterpret, in preoperational terms, some of the existing literature on novice programmers, which has not previously been linked to Piaget.

In section 4, we look at concrete operational reasoning, the middle level of the three Piagetian levels discussed in this paper, which is more abstract than preoperational reasoning but less abstract than formal operational reasoning. The relationship between programming and concrete operational reasoning has not yet been well explored, especially from a neo-Piagetian perspective, and the primary novel contribution of this paper is to explore that relationship.

1.1 Classical Piagetian vs. Neo-Piagetian

Classical Piagetian theory is focussed upon the intellectual development of the child. In Piaget's view, children exhibit increasingly abstract forms of reasoning because of the biological maturation of the brain. Thus, in Piaget's view, a child who exhibits a certain level of abstract reasoning on a given problem will tend to exhibit that same level of abstract reasoning on many other problems. Subsequent work in psychology, however, has shown that children (and adults) exhibit different levels of abstract reasoning on different problems. Some other aspects of classical Piagetian theory have also been cast into doubt (Smith, 1992).

This paper is based upon neo-Piagetian theory. While the types of abstract reasoning are broadly the same in classical and neo-Piagetian theory, the principle difference in neo-Piagetian theory is that people, *regardless of their age*, are thought to progress through increasingly abstract forms of reasoning *as they gain expertise in a specific problem domain*. Thus a person who is a novice in one domain (e.g. chess) will exhibit less abstract forms of reasoning than that same person will exhibit in a domain where he/she is expert (e.g. calculus). For a comprehensive review of neo-Piagetian theories, see Morra *et al.* (2007).

Most neo-Piagetian theorists argue that the increase in abstraction is not a consequence of biological maturity, but instead is due to an increase in the effective capacity of working memory. It is well known that working memory has a very small capacity ─ seven plus or minus two is the popularly known estimate (Miller, 1956). Despite this severe working memory limitation, people can routinely handle more data because of 'chunking'. That is, if a set of associated data items are already stored in long term memory, they may be retrieved and used in working memory as if they were a single item. For example, remembering a new telephone number for a few seconds will consume working memory capacity, as each digit forms one data item to be stored in working memory. However, once a specific phone number has been committed to long term memory, then it only counts as one data item when it is brought back into working memory. Thus, the well known limitations of working memory do not apply to all data, but just to data that has not yet been learnt.

In classical Piagetian theory, it is customary to talk of a person as being in a particular Piagetian developmental stage of reasoning. Many neo-Piagetians (e.g. Biggs and Collis 1982) prefer to describe a person as exhibiting a particular level of abstract reasoning as he/she works on a specific problem – as we will do in this paper.

1.2 The SOLO taxonomy

Our motivation for exploring the implications of neo-Piagetian theory stems from the use of the SOLO taxonomy (Biggs and Collis, 1982) in the BRACElet project. The SOLO taxonomy was inspired by neo-Piagetian theory, so we thought it might be useful to explore the relationship between neo-Piagetian theory and programming.

One of the earliest BRACElet papers to use the SOLO taxonomy (Whalley *et al.*, 2006) reported on the performance of students in an end−of−first−semester exam. As part of that exam, the students were given a question that began "*In plain English, explain what the following segment of Java code does*". Whalley *et al.* found that some students responded with a correct, line−by−line description of the code while other students responded with a correct summary of the overall computation performed by the code (e.g. "*the code checks to see if the elements in the array are sorted*"). A line−by−line description is a SOLO multistructural response, while a correct summary of the overall computation is a SOLO relational response. A relational response is more abstract than a multistructural response.

In another BRACElet study, Lopez *et al.* (2008) analysed the performance of students in an end−of−semester programming exam. They found that the combination of student scores on tracing tasks and "explain in plain English" tasks accounted for 46% of the variance on a code writing task. Three subsequent studies have reported similar results (Lister, Fidge and Teague, 2009; Venables, Tan and Lister, 2009; and Lister *et al.*, 2010).

2 Formal Operational Reasoning

Formal operational reasoning is the most abstract of the Piagetian types of reasoning. We describe this type of reasoning first because (1) it is how expert programmers reason, and (2) in the literature connecting Piaget and programming, formal operational reasoning has received more attention than other forms of Piagetian reasoning.

2.1 General Description

A person reasoning at this level exhibits the thinking characteristics traditionally emphasized at university (or at least we academics like to think so). A person thinking formally can reason logically, consistently and systematically. Piaget nominated sixteen combinatorial outcomes of binary propositions (e.g. "p or q", "p implies q", "p is equivalent to q") that he felt were used in formal reasoning, even when the formal reasoning is expressed implicitly in natural language, and not expressed explicitly in propositional logic. Formal operational reasoning also requires a reflective capacity – the ability to think about one's own thinking. Formal operational thinking can involve reasoning about hypothetical situations, or at least reasoning about situations that have never been directly experienced by the thinker. It also involves an awareness of what is known for certain, and what is known with some probability of being true, which in turn allows someone who is thinking formally to perform hypothetico-deductive reasoning − that is, the making of a tentative inference from incomplete data, then actively, systematically seeking further data to confirm or deny the tentative inference. For a more detailed description of formal operational reasoning, see Brainerd (1978) or Flavell (1977).

2.2 Formal Reasoning in Programming

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 subproblems, (3) transform subproblems into subsolutions, (4) recompose, and (5) evaluate and iterate. Similarly, Fischer (1986) nominated top down design as requiring formal operational reasoning:

The student must be able to see a problem as a statement summarizing a set of inter−related but unstated components, some of which are given, others probable, and still others are merely possible ... one must be able to conceive of all the possible steps or parts of each module or problem, and be able to determine in what order they must occur.

2.2.1 Predictors of Programming Ability

Almost all the literature that connects Piaget and programming does so in the context of attempting to develop a predictor of programming ability, by using a Piagetian test of reasoning. Various authors have reported mixed results:

- Kurtz (1980) constructed a 15 item test of Piagetian reasoning, where each item was taken from a previously published study. He compared the performance of 23 students on that test with their final grade in an introductory programming course, and reported an R^2 of 0.63, which we regard as very high. However ...
- When Barker and Unger (1983) carried out a similar experiment to Kurtz, using 11 of the items from Kurtz's test, their results for a larger population of 353 students "*failed to produce the spectacular ... correlation*" (p. 156) found by Kurtz.
- Fischer (1986) used a previously published test of Piagetian reasoning and found that 91% of students from an introductory programming course who received a course grade of B+ or higher were classified as "formal operational thinkers".
- Cafolla (1988) performed a linear regression between student performance on a previously published test of formal operational reasoning and their performance in the final exam of an introductory programming course. He reported an R^2 of 0.35, which we regard as low.
- As a binary measure of successful/unsuccessful in learning to program, Hudak and Anderson (1990) used a criterion of a final course grade of 80% or higher in a CS1 course. They were able to correctly classify 72% of successful/unsuccessful students, using a previously published test of formal operational reasoning.
- Bennedsen and Caspersen (2006) used a classic Piagetian experiment – the pendulum test – to estimate the formal operational reasoning capacity of programming students, but they found that this estimate did not correlate well with the students' final CS1 grade.

The mixed results reported in the above works may be due to the classical Piagetian perspective adopted in those works. Recall that the neo-Piagetian perspective is that the level of abstract reasoning an individual manifests varies between problem domains. From that neo-Piagetian perspective, there is no reason to expect that a person's ability on a non-programming test of abstract reasoning should correlate with that person's ability at programming.

2.2.2 Grading: Assignments and Exams

In the above reported mixed results, while several of the Piagetian tests used by those authors had been previously validated, the authors made no attempt to validate (and in some cases, did not even describe) the methods used to produce the student grades. Therefore, one possible explanation of the above mixed results is that the grading approaches used are measuring different programming skills, possibly skills at different levels of Piagetian reasoning. (As an aside, we express surprise at the confidence most academics have in their respective grading schemes).

One grading issue is the relative weighting given to assignments and exams. The code students are required to write in assignments is generally more complicated than the code they write in exams. Therefore, in general, the demands placed upon a novice's problem solving skills are greater for assignments than for exam questions – that is, formal operational reasoning may be tested by assignments more than exams.

On the other hand, sometimes a process that combines quasi-random code changes and copious trial runs (a process which is decidedly not formal operational reasoning) can produce a poor but passing assignment solution, while the student writing code in a paper-based exam does not have that luxury. On the other hand again, exam marking can be generous. Sometimes a student's code is well rewarded even though it bears only a superficial appearance to correct code. For example, Traynor, Bergin, and Gibson (2006) provided an illuminating extract from an interview with a student, where the student described his approach to answering coding questions in an exam, when he didn't really know the answer:

... you usually get the marks by making the answer look correct. Like, if it's a searching problem, you put down a loop and you have an array and an if statement. That usually gets you the marks ... not all of them, but definitely a pass".

Dressel (1983) described the grades we give students (in any discipline) as

"... an inadequate report of an inaccurate judgement by a biased and variable judge of the extent to which a student has attained an undefined level of mastery of an unknown proportion of an indefinite material."

To summarize this brief discussion on grading, it is difficult to infer that a student has used formal operational reasoning to produce a piece of code, when the only supporting evidence is the code itself – or worse, a non-validated grade. Formal operational reasoning is more about the mental process rather than the final product.

2.2.3 Testing and Debugging

The process of programming involves a great deal of testing and debugging. As part of advocating that novice programmers should be discouraged from a trial and error approach to programming, and instead encouraged to adopt a reflection−in−action approach, Edwards (2004) recommended that novice programmers needed...

> *"... practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses. ... These activities are at the heart of software testing."* (p. 27)

What Edwards was describing is hypothetico-deductive reasoning, a signature skill of formal operational reasoning.

In a book entitled simply "*Debugging*", Agans (2006) describes a number of approaches to debugging, including "*Make it Fail*" and "*Divide and Conquer*". A programmer who is thinking in a formal operational manner routinely applies general debugging principles like these. Furthermore, while these approaches are applicable to programming, Agans' book is about debugging anything (e.g. an electrical appliance) and thus his book illustrates another aspect of formal operational thinking − the ability to choose the most appropriate abstract concept to use in a specific case.

2.3 A Closing Remark

Formal operational reasoning is how we would like our students to go about writing programs, and how they eventually may go about writing programs, but neo-Piagetian theory tells us that most novices will not immediately begin to think this way about programs. Instead, novices progress toward formal operational reasoning, via less sophisticated forms of reasoning. In the next section, we consider how novices first think about programs.

3 Preoperational Reasoning

Preoperational reasoning is the least abstract form of Piagetian reasoning discussed in this paper. From a neo-Piagetian perspective, most novices in any problem domain begin with this type of reasoning. Therefore, even though some novice programmers may progress quickly to more sophisticated forms of reasoning, we should expect to see preoperational reasoning from most novices when they first begin to program.

3.1 General Description

A person who is thinking in a preoperational mode tends not to form many abstractions from the objects in the problem environment. Their thinking reflects the direct manipulations they could make to that environment. There is little thinking about the relationships between the objects. To the limited extent that preoperational thinking abstracts beyond the actual objects, the thinking is not systematic. Also, the thinking tends to focus on only one abstract property at any given moment in time, and when more than one abstract thought occurs over time those abstractions are not coordinated, and may be contradictory. For a more detailed description of classical preoperational reasoning, see Brainerd (1978) or Flavell (1977).

From the neo-Piagetian perspective, the low level of abstraction in preoperational thinking, on a particular problem, is a consequence of the novice's working memory being overwhelmed, since the novice has not yet learnt to chunk knowledge and information in that problem domain.

3.2 Preoperational Reasoning in Programming

In this section, we will describe novice programmer behaviours that are the staple diet of conversations among academics who teach programming – often exasperated and incredulous conversations.

An attractive quality of neo-Piagetian theory is that it makes dealing with these behaviours less exasperating for the teacher. Neo-Piagetian theory allows us to see that these behaviours are not the manifestation of cognitive dysfunction by a student, nor are the behaviours due to mental laziness. Instead these behaviours are a normal stage of cognitive development.

3.2.1 Tracing Without Abstracting Meaning

Preoperational students can trace code. That is, they can manually execute a piece of code and determine the values in the variables when the execution is finished. Research suggests that novices need to be able to trace with >50% accuracy before they can begin to understand code (Philpott, Robbins and Whalley, 2007; Lister, Fidge and Teague, 2009; Venables, Tan and Lister, 2009). Students who trace code with less than 50% accuracy are operating at a lower Piagetian level than preoperational, which is called the sensorimotor level, but we do not discuss that level any further in this paper.

A defining characteristic of preoperational reasoning in programming is that, while such a novice can reliably trace code, that novice does not routinely abstract from the code to see a meaningful computation performed by that code. If pressed by their teacher to offer a meaningful computation performed by a piece of code, the novice who is thinking preoperationally may make a reasonable inductive guess, based upon the input/output behaviour they observe from tracing the code, but that novice will not infer the computation deductively, from the code itself. For the novice who is thinking preoperationally, the lines in a piece of code are only weakly related.

The ITiCSE 2004 "Leeds" working group (Lister *et al.*, 2004) collected data from some end−of−first−semester students, using a think−out−loud protocol. Eight of those students could trace code reliably, and answer questions about what values were in the variables after the code had finished executing, but...

> *"... While working out their answer, none of these students volunteered any realization of the intent of the code, to count the number of identical elements in the two arrays ..."* (p. 138)

In contrast, when Lister *et al.* (2006) gave the same problem to several expert programmers, those experts tended to avoid tracing. Instead the experts first read the code to deduce what it did – the code, as indicated in the above quote, counted the number of identical elements in two sorted arrays. Having deduced what the code did, the experts then simply counted the number of common elements in the two arrays, and did not hand execute the code.

3.2.2 Diagrams

Since novice programmers who reason preoperationally tend not to abstract, and when they do abstract they tend to not be systematic, these novices struggle to make effective use of diagrammatic abstractions of code. Thomas, Ratcliffe, and Thomasson (2004) wrote despairingly of their frustrations at trying to get their novices to make effective use of diagrams:

> *... when they might appropriately use* [diagrams] *themselves, weaker students fail to do so. They are often impatient when the instructor resorts to drawing a diagram, then amazed that the approach works. ...* [also] *providing* [students] *with what we considered to be helpful diagrams did not significantly appear to improve their understanding .... This was completely unexpected. We thought that we were 'practically doing the question for them'...*

Lister (2007) reported a related experience. He identified a group of students in his class who were adept at tracing code, but who could not make use of diagrams. Specifically, in the end−of−first−semester exam, he provided the students with code that implemented algorithms that the class had studied during the semester. One or two lines of code were omitted from each piece of code, and the students had to choose the missing lines in multiple choice questions. He provided diagrams of each algorithm tested, but the "middle novice programmers" (as he called them in the paper) struggled to use the diagrams to choose the correct answer.

3.2.2.1 Doodling

The ITiCSE 2004 "Leeds" working group (Lister *et al.*, 2004) collected "doodles" from their end−of−first−semester students. Doodles are, as Lister *et al.* defined the term, the diagrams and other annotations that programmers make as they reason about a piece of code. They looked at the types of doodles generated by their novices on two questions. For one of those questions, one fifth of the students made no doodles at all. For the other question, more than one half of the students made no doodles at all. Lister *et al.* were surprised, but their finding makes sense from a neo-Piagetian perspective, since novices who are reasoning preoperationally lack the mental abstractions to make doodling useful to them.

3.2.3 Code Explanation

The neo-Piagetian perspective, and in particular what is known about preoperational reasoning, provides an explanation as to why the BRACElet project has found (as reported in the introduction) that some students struggle with "explain in plain English" tasks. The explanation is based around the neo-Piagetian idea that the working memory of these novices is easily overloaded, as they lack the knowledge structures to "chunk" the code.

First, let us consider a particularly easy piece of code to understand, which increments all elements of an array:

```
for (int i=0 ; i<x.length ; ++i )
   x[i] = x[i] + 1;
```

The preoperational novice need only understand two things about that code: (1) that the variable "i" will take on a set of values which map to all elements of the array "x", and (2) the single line of code in the body of the loop will increment an element of the array. This code has two critical features that make it an example of the simplest type of iterative/array code that a novice can be called upon to understand, because: (1) each iteration of the loop performs the same process as the other iterations, but each iteration performs that process on a unique element of the array, and (2) no iteration affects what happens on any other iteration. This code is an example of iterative/array code that places low demands on the novice's working memory.

Second, consider the code below, which sums the elements of an array:

```
int sum = 0;
for (int i=0 ; i<x.length ; ++i )
     sum = sum + x[i];
```

As in the first example, the preoperational novice needs to understand that the variable "i" will take on a set of values which map to all elements of the array "x". The remaining code in this second example is slightly harder to grasp than the code in the first example. The novice must understand that the single line of code in the body of the array will accumulate values in a single variable, "sum". However, this second example is still relatively simple code to understand because, like the first example, each iteration of the loop performs the same process. There is an additional burden on the novice in this second example, and that is the line of code that initializes "sum". In total, however, this second example is only slightly harder for the novice to understand than the first example.

Third, and finally, consider the classic BRACElet "explain in plain English" problem (Whalley *et al.*, 2006; Lister *et al.*, 2006), which is shown below:

```
bool bValid = true;

for (int i = 0 ; i < iMAX-1 ; i++)
{
   if (iNumbers[i] > iNumbers[i+1])
          bValid = false;
}
```

That code checks to see if the array is sorted. As reported in several BRACElet papers, students have some difficulty with explaining this code. There are two aspects of this code that make it more difficult to understand for the novice programmer: (1) the novice needs to reason about two different array elements in each loop iteration, and that (2) once the "bValid" variable changes its value, it cannot get its original value back again.

There is another factor in why pre-operational novices find it hard to explain the above classic BRACElet question – they lack the capacity for *transitive inference*. That type of inference is a form of concrete operational reasoning, which we describe and discuss later in this paper.

3.3 A Closing Remark

Programming teachers want their students to develop beyond these preoperational behaviours as quickly as possible, but currently many of our students do not develop beyond these behaviours. While this lack of development may be a failing of some of the students, it may also be due to our existing pedagogical practices. We teachers admonish students for their preoperational behaviours, but we do not offer them learning experiences targeted at moving them beyond these preoperational behaviours. By the end of the next section of the paper, on concrete operational reasoning, we will have identified learning and assessment activities that could be used to encourage the novice to move beyond preoperational reasoning.

4 Concrete Operational Reasoning

Concrete operational reasoning is the middle level of the three Piagetian levels discussed in this paper − more abstract than preoperational reasoning but less abstract than formal operational reasoning.

4.1 General Description

As is often the case with intermediate levels in many hierarchies, concrete thinking is frequently defined in terms of how it differs from the other two levels of Piagetian thinking. Unlike preoperational thinking, concrete thinking does involve routine reasoning about abstractions from the objects in the environment. However, a defining characteristic of concrete thinking is that the abstract thinking is restricted to familiar, real situations, not hypothetical situations (hence the name "concrete"). Consequently, the hypothetico-deductive reasoning of formal operational reasoning tends not to be manifested in concrete reasoning. For a more detailed description of classical concrete operational reasoning, see Brainerd (1978) or Flavell (1977).

4.1.1 Conservation and Reversing

The archetypal manifestation of concrete thinking is the ability to reason about quantities that are conserved, and processes that are reversible. In Piaget's own work, the famous illustration of concrete reasoning is the conservation of liquid task. This task involves three glasses, two of which are identical in shape, with both of those glasses initially containing an equal amount of water. When studying children, Piaget would (1) ask the child to agree that the two identical glasses contained the same amount of water; (2) pour the water in one of the glasses into the empty third glass, which had a different width from the other two glasses; and finally (3) Piaget would ask the child if the third glass contained less, more, or the same amount of water as the other glass that contained water. Younger children, who are thinking in a preoperational mode, will give different answers depending on the height of the water in the two glasses. Older children (and adults) who are thinking in a concrete mode will immediately claim that the amount of liquid is the same in both glasses. If pressed to justify that claim, the child (or adult) may argue that, although the liquid is at different heights in the two containers, the effect of the differing container widths compensates for the differing heights. If pressed further, the child (or adult) may argue that if the liquid was poured back into the glass from which it came, then the liquid will reach the same level as it did initially.

Before dismissing the incorrect preoperational reasoning of a small child on this conservation of liquid task as merely being a manifestation of a biologically immature mind, the reader might consider the more sophisticated but incorrect "commonsense beliefs" of adults. For example, research has demonstrated that many university students who commence study in physics have "commonsense beliefs" about motion and force that are incompatible with Newton (Halloun and Hestenes, 1985). A child's incorrect thinking on the conservation of liquid task highlights the same sort of incorrect thinking that adults can also make, but on more sophisticated tasks. (Indeed, Piaget saw the development of thinking in children as a recapitulation of the historical development of scientific thinking.) Later in this section of the paper on concrete operational thinking (i.e. in 4.2.3), we will nominate programming tasks that also involve the principles of reversibility and conservation.

The neo-Piagetian explanation for the conservation of liquid task, which generalizes to other conservation tasks, is as follows. A child who reasons preoperationally (and incorrectly) does so because of their limited working memory capacity. That limited capacity only allows such a child to focus on a single measure of the quantity of liquid in a glass – the height of the water. A child (or adult) who reasons concretely (and correctly) first begins to do so when an increase in working memory capacity (due to chunking) allows them to consider two measures of the quantity of liquid in a glass – both the height and the width of the water. When a child is able to simultaneously appreciate the relationship between height and width, the child is then open to learning that the effect of height and width can sometimes cancel each other out, and from there the child can learn the more abstract concept that the quantity of liquid is conserved.

4.1.2 Transitive Inference

Transitive inference is another important characteristic of concrete operational reasoning. It is the type of reasoning where, in general terms, if a certain relationship holds between object A and object B, and if the same relationship holds between object B and object C, then the same relationship also holds between object A and object C. For example, Piaget would sometimes ask a child a question like, "*If Adam is taller than Bob, and Bob is taller than Charlie, who is the tallest*?"

From the neo-Piagetian perspective, the reason why a novice may not be able to perform transitive inference is that the working memory of the novice is overloaded by other information, because the novice has not yet learnt to chunk information in this problem domain. Consequently, the novice cannot hold simultaneously in working memory all the information about the relationships between A, B and C required to perform the transitive inference.

4.2 Concrete Reasoning in Programming

There is small amount of classical Piagetian literature on concrete operational reasoning in programming, which describes how young children learn to program (e.g.

Huber, 1985). However, we are not aware of any literature that explicitly connects programming with neo-Piagetian literature on concrete operational reasoning. There is some literature, however, that does describe some of the behaviours of novice programmers that are concrete operational, without making the connection to neo-Piagetian literature. In the next two subsections on concrete reasoning, we describe two examples of that literature.

4.2.1 Ginat: Hasty design, futile patching

Recall from the general description of concrete operational reasoning that any reasoning about abstractions is restricted to familiar, real situations, and that hypothetico-deductive reasoning is not part of concrete operational reasoning. Ginat (2007) observed concrete reasoning, with the absence of hypothetico-deductive reasoning, in novice programmers:

> *A hasty design may be based on some simplistic application of a familiar design pattern ... The design pattern's invocation may be relevant. Yet, its utilization may not be based on sufficient task analysis and thorough examination of diverse input cases, but rather on some premature association that seems relevant. Errors are not always discovered, as the test cases on which the program is tested are very limited. The devised program is batched, and "seems correct". Then, an outside source (e.g., a teacher) points out a falsifying input. A patch is offered. Sometimes the patch is sufficient for yielding correctness, but more often than not, the patch is insufficient. An additional patch is offered; and the cycle of batch−&−patch continues.*

The following quotation from Flavell, Miller and Miller (2002) is about concrete reasoning in children. We highlight the similarity between what Flavell *et al.* wrote, and what Ginat wrote, by striking through Flavell *et al.*'s references to the elementary school child and replacing it with references to the novice programmer:

> *The* [concrete operational novice programmer's] ~~*elementary school child's*~~ *characteristic approach to many conceptual problems is to burrow right into the problem data as quickly as possible.... His is an earthbound, concrete, practical minded sort of problem solving approach, one that persistently fixates on the perceptible and inferable reality right there in front of him. His conceptual approach is definitely not unintelligent and it certainly generates solution attempts that are more rational and task−relevant than the preoperational* [novice programmer] ~~*child*~~ *is likely to produce. It does, however, hug the ground of detected empirical reality rather closely, and speculations about other possibilities ... occur only with difficulty and as a last resort. An ivory−tower theorist the* [concrete operational novice programmer] ~~*elementary school child*~~ *is not. ... For the concrete operational thinker, the realm of the abstract possibility is seen as an uncertain and only occasional extension of the safer and surer realm of palpable reality* (p. 146)

The purpose in providing the above quote is not to suggest that novice programmers behave like school children. Instead, the purpose is to highlight the behaviour of novices, at any age, in any problem domain, including programming, when their reasoning is concrete operational.

4.2.2 Hazzan: Reducing Abstraction

Although Hazzan (2008) did not describe her work in neo-Piagetian terms, she described how novices simplify programming tasks from formal operational to concrete operational reasoning:

> *... students, when facing the need to cope meaningfully with concepts that are too abstract for them, tend to reduce the level of abstraction in order to make these abstract concepts meaningful and mentally accessible ... by dealing with specific examples instead of with a whole set defined in general terms.*

4.2.3 Concrete Operational Tasks

In the remainder of this section on concrete operational reasoning, we propose and explore some examples of formative and/or summative tasks that could be given to students to develop and/or test their concrete operational reasoning.

4.2.3.1 <u>Reversing</u>

As discussed in the general description of concrete operational reasoning, the archetypal manifestation of concrete thinking is the ability to reason about quantities that are conserved, and processes that are reversible. What follows is a task that requires the student to reason about reversing.

> The purpose of the following code is to move all elements of the array x one place to the ***right***, with the ***rightmost*** element being moved to the ***leftmost*** position:
>
> ```
> int temp = x[x.length-1];
>
> for (int i = x.length-2; i>=0; --i)
>     x[i+1] = x[i];
>
> x[0] = temp;
> ```
>
> Write code that undoes the effect of the above code. That is, write code to move all elements of the array x one place to the ***left***, with the ***leftmost*** element being moved to the ***rightmost*** position.

An important feature of a solution written by someone using concrete reasoning is the reversal of the direction of the loop. A novice who reasoned preoperationally might miss that change, at least until that novice had a chance to run their initial solution and discover the error.

Note, however, that we cannot conclude that a novice has performed concrete operational reasoning if the only evidence we have is the novice's final, correct solution, and the novice used a computer. A novice might solve this reversing problem by a process that combines quasi-random code changes and copious trial runs, which is the behaviour of someone reasoning preoperationally. In contrast, the novice who solves this problem by applying concrete operational reasoning will, after inspecting the given code, produce a correct solution almost immediately (apart, perhaps, from trivial syntax errors). To be confident that the novice performed concrete operational reasoning, we must either: (1) observe the novice as they work on the problem, (2) use software to monitor, or limit, the number of edit−compile−run cycles, or (3) have the student write the solution on paper.

4.2.3.2 Conservation

Cases of conservation in programming systems are more abstract than conservation in physical systems, such as the conservation of the volume of a liquid as it is poured from one container to another. One case in programming is the conservation of a specification when the underlying implementation is changed. Consider the following question:

Below is incomplete code for a method which returns the smallest value in the array “x”. The code scans across the array, using the variable “minsofar” to remember the smallest value seen thus far. There are two ways to implement remembering the smallest value seen thus far: (1) remember the actual value, or (2) remember the value’s position in the array. Each box below contains two lines of code, one for implementation (1), the other for implementation (2). First, make a choice about which implementation you will use (it doesn’t matter which). Then, for each box, draw a circle around the appropriate line of code so that the method will correctly return the smallest value in the array.

```
public int min( int x[] ){

                  +-------------------+
int minsofar =    | (a) 0             |  ;
                  | (b) x[0]          |
                  +-------------------+

for ( int i=1 ; i<x.length ; ++i )
{                  +------------------+
    if ( x[i] <    | (c) minsofar     |  )
                   | (d) x[minsofar]  |
                   +------------------+

                      +--------------+
       minsofar =     | (e) i        |  ;
                      | (f) x[i]     |
}                     +--------------+
           +-------------------+
return     | (g) minsofar      |  ;
           | (h) x[minsofar]   |
}          +-------------------+
```

An alternative version of this question would provide the code for one of the implementations, and then ask the student to provide the other implementation.

The ability to make simple transformations between implementations, while conserving the specification, is an underappreciated skill in programming pedagogy. The novice who can easily perform simple implementation transformations is then less preoccupied by implementation minutiae, and can then focus upon higher abstractions about a program. Current pedagogical practice, however, does not ensure that novices have this concrete operational skill before giving them tasks that require formal operational reasoning.

4.2.3.3 Transitive Inference (1)

Consider the following question:

In one sentence, describe the purpose of the following code. Assume that the variables y1, y2 and y3 contain integer values. In each of the three boxes that contain sentences beginning “Swap the values in …” assume that appropriate code is provided – do NOT write that code.

```
if (y1 < y2)
     +----------------------------------+
     | Swap the values in y1 and y2.    |
     +----------------------------------+
if (y2 < y3)
     +----------------------------------+
     | Swap the values in y2 and y3.    |
     +----------------------------------+
if (y1 < y2)
     +----------------------------------+
     | Swap the values in y1 and y2.    |
     +----------------------------------+
```

A suitable answer to this question would be “*It sorts the three values so that* y1 ≥ y2 ≥ y3”. To make such a deduction, the novice must perform transitive inference.

While some readers may at first be sceptical that any novice could have difficulty answering the above question, we report in another paper in these same proceedings (Corney, Lister and Teague, 2011) how we used this question in a class test, in the fifth week of an introductory course, and found that half of our students could not answer the question.

4.2.3.4 Transitive Inference (2)

Here again is the code for the classic BRACElet “explain in plain English” problem (Whalley *et al.*, 2006; Lister *et al.*, 2006):

```
bool bValid = true;

for (int i = 0; i < iMAX-1; i++)
{
   if (iNumbers[i] > iNumbers[i+1])
           bValid = false;
}
```

If a novice is to offer an explanation like “*It checks to see if the array is sorted*”, then the novice must perform transitive inference. That is, the novice must recognize (albeit not necessarily in the mathematical notation used here) that if `bValid` is still true at a given value of `i`, then *for all* `p,q` such that `0 ≤ p < q ≤ i+1` ...

`iNumbers[p] ≤ iNumbers[q]`

4.3 A Closing Remark

Today, not only do we expect programming students to move beyond preoperational reasoning quickly, but we also expect them to go directly from preoperational reasoning to formal operational reasoning. However, Piagetian theory indicates that such a transition is difficult, that instead novices tend to move to formal operational reasoning via the concrete operational level. The primary weakness of today's pedagogy of programming, with its heavy emphasis on writing large amounts of code, on problem solving and on top down design, is that it doesn't provide an opportunity for the novice to develop concrete operational skills, via the types of exercises we have described in this section of the paper. Consequently, the novice who is unable to bridge for themself the gap between preoperational reasoning and formal operational reasoning remains stuck in preoperational reasoning.

5 Conclusion

In the CS1 classroom, our students can exhibit three broad forms of neo-Piagetian reasoning. When we lecture to our class, we tend to talk about programs in terms of formal operational reasoning, the most abstract of the three forms of neo-Piagetian reasoning – it is only natural that we would do so, since we who lecture are accomplished programmers. However, many of our students have not yet reached a point (not in CS1) where they comfortably follow a discussion of a program in formal terms – for those students, we may as well be lecturing in a foreign language, and in a sense we are.

Some of our students tend to reason in a preoperational form, where they can trace the changing values in specific code, but do not reason in terms of abstractions of the code. Some other students tend to reason in a concrete operational form, where they can see some abstractions of specific code, but they can only see those abstractions in the context of that specific code. Most students who tend to reason at the preoperational and concrete operational levels will not spontaneously become more accomplished at the formal operational level simply by being exposed to learning materials couched in formal operational terms.

It is often said CS1 students need to practice more, by writing more programs. Students who tend to reason formally about code, and perhaps also students who tend to reason concretely about code, may indeed benefit from writing more code. However, students who tend to reason preoperationally about code will gain little from being forced to write large quantities of code. Such students can only write code by quasi-random mutation. For students who are predominately reasoning at the preoperational level, and perhaps also for students who are predominately reasoning at the concrete level, we need to develop new types of learning experiences that develop their abstract reasoning without requiring them to write a lot of code.

In computing, there has long been a debate about whether programming skill is innate, or acquired. The neo-Piagetian perspective may afford a means of transcending that dialectic. There may indeed be people who will never learn to reason in a formal operational way about programs, but today's high failure rates in introductory programming courses probably overstates the percentage of people who could never learn to program. Just because a particular individual does not spontaneously manifest formal operational reasoning about programs does not mean that the individual cannot possibly learn to do so, given the right tuition. Perhaps, with a pedagogical approach informed by neo-Piagetian theory, a pedagogical approach that recognizes preoperational and concrete operational reasoning as legitimate developmental phases, which works at increasing the sophistication of how a student reasons about code, via learning experiences that do not require the student to write copious quantities of code, then perhaps such a student can learn to reason in a formal operational way about programs.

While students who aspire to be professional software engineers must eventually develop formal operational programming skills, perhaps it is unrealistic to expect most of those students to do so in their first semester of programming. Perhaps, in that first semester, we teachers should set our sights on getting the bulk of our students to the point where they can consistently reason at the concrete operational level.

Acknowledgements

The author's work was partially funded by an Associate Fellowship awarded by the Australian Learning and Teaching Council. The author's thinking about neo-Piagetian theory stems from the empirical results of the BRACElet project. He thanks his many collaborators on that project, especially Tony Clear and Jacqui Whalley.

References

Agans, D. (2006) Debugging New York: Amacom

Barker, R. and Unger, E. (1983) *A predictor for success in an introductory programming class based upon abstract reasoning development. SIGCSE Bull.* 15, 1 (Feb.), 154-158. DOI=10.1145/952978.801037

Bennedsen, J. and Caspersen, M. (2006) *Abstraction ability as an indicator of success for learning object-oriented programming*? *SIGCSE Bull.* 38, 2 (June), 39-43. http://doi.acm.org/10.1145/1138403.1138430

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

Brainerd, C. (1978) *Piaget's theory of intelligence*. Englewood Cliffs, N.J.: Prentice-Hall.

Cafolla, R. (1988) Piagetian Formal Operations and Other Cognitive Correlates of Achievement in Computer Programming. *Journal of Educational Technology Systems,* vol. 16, no. 1, p45-55.

Corney, M., Lister, R. and Teague, D. (2011) *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. The 13th Australasian Computer Education Conference (ACE 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 114. J. Hamer and M. de Raadt, Eds. http://crpit.com/index.html

Dijkstra, E. 1979. *The humble programmer*. In Classics in Software Engineering, E. N. Yourdon, Ed. ACM Classic Books Series. Yourdon Press, Upper Saddle River, NJ, 111-125.

Dressel, P. (1983) *Grades: One More Tilt at the Windmill*, in A. W. Chickering (ed.), Bulletin, Memphis State University, Center for the Study of Higher Education, Memphis, December.

Edwards, S. (2004) Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bull.* 36, 1 (March), 26-30. http://doi.acm.org/10.1145/1028174.971312

Fischer, G. (1986) *Computer Programming: A Formal Operational Task.* 16th Annual Symposium of the Piaget Society, Philadelphia, PA, USA. http://www.eric.ed.gov/PDFS/ED275316.pdf

Flavell, J. (1977) *Cognitive development*. Englewood Cliffs, N.J.: Prentice-Hall

Flavell, J., Miller, P. and Miller, A. (2002) *Cognitive development.* (4th edition) Upper Saddle River, N.J.: Prentice-Hall.

Ginat, D. (2007). Hasty design, futile patching and the elaboration of rigor. *SIGCSE Bull.* 39, 3 (June), 161-165. http://doi.acm.org/10.1145/1269900.1268832

Halloun, I. and Hestenes, D. (1985) The initial knowledge state of college physics students, *Am. J. Phys*. 53(11), 1043-1055.

Hazzan, O. (2008). Reflections on teaching abstraction and other soft ideas. *SIGCSE Bull.* 40, 2 (June), 40-43. http://doi.acm.org/10.1145/1383602.1383631

Huber, L. (1985) Computer Learning Through Piaget's Eyes. *Classroom Computer Learning*, Vol. 6, No. 2 (Oct), pp. 39-43.

Hudak, M., and Anderson, D. (1990) Formal Operations and Learning Style Predict Success in Statistics and Computer Science Courses. *Teaching of Psychology*, 17(4) 231-234.

Kramer, J. (2007) Is abstraction the key to computing? *Communications of the ACM,* Vol. 50, 4 (April), 36-42. http://doi.acm.org/10.1145/1232743.1232745

Kurtz, B. (1980) Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *SIGCSE Bull.* 12(1), 110-117. http://doi.acm.org/10.1145/953032.804622

Lister R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004) A Multi-National Study of Reading and Tracing Skills in Novice Programmers. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.* 36, 4 (June), 119-150. http://doi.acm.org/10.1145/1041624.1041673

Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006) Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.* 38, 3 (June), 118-122. http://doi.acm.org/10.1145/1140123.1140157

Lister, R. (2007): *The Neglected Middle Novice Programmer: Reading and Writing without Abstracting*. 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Nelson, New Zealand, Mann, S. and Bridgeman, N., Eds, 133-140. http://www.naccq.ac.nz/conferences/2007/133.pdf

Lister, R., Fidge C. and Teague, D. (2009) Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *SIGCSE Bull.* 41, 3 (July), 161-165. http://doi.acm.org/10.1145/1595496.1562930

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

Lopez, M., Whalley, J., Robbins P. and Lister, R. (2008) *Relationships between reading, tracing and writing skills in introductory programming. Fourth international Workshop on Computing Education Research* (ICER). pp. 101-112. http://doi.acm.org/10.1145/1404520.1404531

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). pp 125-140.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63, 81–97.

Morra, S., Gobbo, C., Marini, Z. and Sheese, R. (2007) *Cognitive Development: Neo-Piagetian Perspectives*. Psychology Press.

Philpott, A, Robbins, P., and Whalley, J. (2007) *Accessing the Steps on the Road to Relational Thinking*. 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Port Nelson, New Zealand, Mann, S. and Bridgeman, N., Eds, p. 286.

Smith, L. (Ed.) (1992) *Jean Piaget : critical assessments*. London ; New York : Routledge.

Thomas, L., Ratcliffe, M., and Thomasson, B. (2004 Scaffolding with object diagrams in first year programming classes: some unexpected results. *SIGCSE Bull.* 36, 1 (March), 250-254. http://doi.acm.org/10.1145/1028174.971390

Traynor, D., Bergin, S., and Gibson, J. P. (2006): *Automated assessment in CS1*. 8th Australian Conference on Computing Education (ACE), Hobart, Australia, ACM International Conference Proceeding Series, 165: 223-228. http://crpit.com/confpapers/CRPITV52Traynor.pdf

Turner, A. (1991) Computing Curricula 1991. *Communications of the ACM*, 34(6), pp. 68-84.

Venables, A., Tan, G. and Lister, R. (2009) *A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer.* Fifth International Workshop on Computing Education Research (ICER). pp. 117-128. http://doi.acm.org/10.1145/1584322.1584336

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*. 8th Australian Conference on Computing Education (ACE), Hobart, Australia, ACM International Conference Proceeding Series, 165: 243-252. http://crpit.com/abstracts/CRPITV52Whalley.html