Neo-Piagetian Theory and the Novice Programmer

Donna Teague

Faculty of Science and Engineering
Queensland University of Technology
d.teague@qut.edu.au

Keywords: POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-V.B. protocol analysis, POP-VI.E. computer science education research

Abstract

This PhD research draws on neo-Piagetian theories of cognitive development to explain how novices learn to program. From an interpretive perspective, we used think aloud studies to observe novice programmers completing simple programming tasks in order to determine the reasoning skills they had utilised. The concurrent verbal reports from the think aloud studies are triangulated with in-class test data of large student cohorts to maximise the quality of the research data and validity of the results. The outcome of the research will be a mapping of the neo-Piagetian stages to behaviours exhibited by novice programmers.

1. Motivation

Why do so many students find programming hard to learn?

One might define an "expert programmer" (borrowing from Bloom's taxonomy) as someone who is able to remember, understand and apply programming concepts, analyse and evaluate programs, and ultimately create their own (Krathwohl, 2002). Expert programmers exhibit a high level of abstract reasoning, and we need to know how that reasoning is developed in order to influence the transition of students from one stage to the next more complex level of cognition in that domain. The results of this study will have pedagogical implications.

2. Background

Piagetian-based cognitive development theories (Flavell, 1977; Morra, Gobbo, Marini, & Sheese, 2007; Piaget, 1952) provide a framework for describing the domain-specific development of cognition. There has been some work in the mathematics domain (Keats, Collis, & Halford, 1978; Ojose, 2008) and also in identifying misconceptions with programming concepts (Clancy, 2004; du Boulay, 1989; Pea, 1986) which help us to understand the problems students have with programming. However, little is known about when, why and how abstract reasoning skills are developed in the programming domain.

It has been hypothesised (Lister, 2011) that programming students exhibit characteristics at each of the sensorimotor, preoperational, concrete operational and formal operational levels described in neo-Piagetian theory. If so, this would explain why some novice programmers struggle with programming, because neo-Piagetian theory considers adequate exposure to the domain of knowledge as paramount to the progression to the next more complex level of abstract reasoning.

At the least most mature stage of cognitive development, sensorimotor, a novice programmer has difficulty tracing (hand executing) code and has a fragile model of program execution. At the next more mature stage, a preoperational novice has overcome any early misconceptions and can now trace code with some accuracy. However, a preoperational programmer is as yet unable to reason about the code's purpose because they are also unable to see any relationship between various parts of the code. They rely on specific values and inductive inference, based on input/output pairs, to determine the outcome of code. A concrete operational programmer, however, is able to reason about code's purpose simply by reading it, as they are able to conceive the unified whole, rather than just a collection of parts. A defining characteristic of the concrete operational stage is the ability to reason about concepts of conservation, reversibility and transitive inference. By the time a programmer reaches the formal operational stage, they can reason logically, consistently and systematically about code. They can understand and use abstractions to reason about and create complex programs of their own.

We each progress through the stages at our own rate. So by virtue of our current teaching practices, students who develop slowly through the early stages would likely be expected by their teachers to operate at a more mature level of reasoning before they are developmentally able.

3. Research Questions

In order to operationalise the question:

Why do so many students find programming hard to learn?

We decompose it into the following questions:

1. Over time, do students tend to exhibit characteristics of each of the neo-Piagetian stages in order from least to most mature?
2. If a student is found to exhibit characteristics of one of the neo-Piagetian stages on a particular programming problem, does that student tend to manifest that same neo-Piagetian stage on similar programming problems?
3. Does a student's programming ability improve with progression into the next more mature neo-Piagetian stage?

4. Objectives

The main objectives of this research project are to:

1. document the reasoning skills and behaviours of novice programmers;
2. analyse the manifestation of those behaviours using neo-Piagetian cognitive development theory; and
3. develop a mapping between neo-Piagetian cognitive development stages and programming reasoning skills and behaviour

The expected outcome of this research is the formalisation of the development trajectory from novice programmer in terms of the evolution of reasoning skills. The formalisation will include behaviours likely to be exhibited, and artefacts likely to be produced by a programmer at each stage of development. Should it be found that novices develop programming skills according to the sequential, cumulative neo-Piagetian stages of development, we can then talk about the "when" of novices' ability becoming expert, rather than "if". It will mean that, given time, everyone can learn to program.

The pedagogical significance of this research is in providing the framework upon which to identify students' cognitive development in programming based on their abstract reasoning behaviour. Novices' learning can then be supported and scaffolded in a stage-appropriate manner in order to better influence their progression to the next more complex level of cognitive development in programming.

5. Method

Normally the only true artefacts we have of our programming students' work are their exam scripts. That is, the exam paper on which they write their solutions to the exam questions. Many assumptions are made by academics about the reasoning employed by students to produce their solutions. From an exam script, we actually know very little about the thought processes, problem solving and reasoning skills the students employed. Likewise, we can only guess about any misconceptions they may have had. In order to gather that sort of rich data, it is necessary to observe the phenomenon as it occurs.

This research uses microgenetic analysis to closely observe novice programmers as they complete programming exercises. The microgenetic research method has been used in other domains to study cognitive development and has been defined as having three key characteristics (Siegler & Crowley, 1991):

1. observations span a period of rapidly changing competence;
2. the density of observations is high relative to the rate of change in competence; and
3. observations are subjected to intensive analysis, with the goal of inferring the processes that gave rise to the change.

The microgenetic research for this project involved interviews with novice programmers. Each interview session was audio-taped, and the participants use a SmartPen (LiveScribe, 2014) and dot paper to complete each exercise while thinking aloud (Ericsson & Simon, 1993). The SmartPen produces a replayable "pencast" which is then able to be encoded for analysis.

Data was also collected from in-class tests and final exams of entire cohorts of novice programmers using the same or similar programming exercises as used in the think aloud studies. Because of this, we can employ triangulation of the qualitative and quantitative methods to check the validity of our findings. From the in-class test artefacts we can make generalisations about the entire cohort, and any problems/misconceptions identified here have informed a specific investigation with the think aloud studies. Conversely, the interesting behaviour or misconceptions observed in the think aloud sessions have lead us to deploy appropriate tests to the entire cohort in order to identify patterns and test research theories or assumptions.

In our final analysis, think aloud students' performance will be categorised according to the skills exhibited while processing tasks associated with certain levels of abstract reasoning. Analysis of the verbal reports will allow quantification of certain behavioural components for each of the students, and allow comparison of students operating at the same and different levels of reasoning.

6. Results to Date

We designed programming tasks which tested for the existence of reasoning skills described in neo-Piagetian theory and then gave them to introductory programming students in the classroom ("in-class tests"). We found evidence that many students continued to struggle with very simple programming tasks which tested them at the concrete operational level (Teague, Corney, Fidge, et al., 2012). The data we have collected from these in-class tests, as well as final exams, supports our initial claim (Corney, Lister, & Teague, 2011) that the problems some students face in learning to program start very early in the semester (Teague, Corney, Ahadi, & Lister, 2012).

We then presented empirical results (Corney, Teague, Ahadi, & Lister, 2012) in support of the neo-Piagetian perspective that novice programmers pass through at least three stages: *sensorimotor*, *preoperational*, and *concrete operational* stages, before eventually reaching programming competence at the *formal operational* stage. The programming exercises we gave novices tested for the concrete operational abilities to reason with quantities that are conserved, processes that are reversible and properties that hold under transitive inference. Examples of each of these types of exercises are shown in Figure 1, Figure 2 and Figure 3. The empirical results from these tests demonstrate that many students struggle to answer these problems, despite their apparent simplicity.

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[] ){
                    ┌──────────────┐
                    │ (a) 0        │
 int minsofar =     │ (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]   │
}             └───────────────────┘
```

Figure 1: Test of Conservation

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.

Figure 2: Test of Reversibility

In plain English, explain what the following segment of Java code does:

```
bool bValid = true;

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

     bValid = false;

}
```

Figure 3: Test of Transitive Inference

From our observational studies, we have pencasts of students at various levels of competency, completing exercises similar to those shown in Figure 1, Figure 2 and Figure 3 which tested their level of abstract reasoning. These think aloud sessions confirmed that students can still be at the sensorimotor and preoperational stages even after two semesters of learning to program (Teague, Corney, Ahadi, & Lister, 2013). These results are the first observational data that is described explicitly in neo-Piagetian terms. Further microgenetic research have provided evidence of novice programmers' evolving ability to reason abstractly which has been analysed using the neo-Piagetian framework (Teague & Lister, 2014a, 2014d).

What has emerged from the think aloud studies is evidence for three different ways in which students reason about programming which correspond to the first three neo-Piagetian stages (Lister, 2011). At the sensorimotor stage, novices programmers exhibit misconceptions and other errors that are already well established in the literature (e.g., Du Boulay (1989)). At the next stage, known as the preoperational stage, students can correctly trace a program, but they can neither reason about code nor see a relationship between parts of a program. Preoperational programming students are not yet equipped with skills which allow them to reason about conservation, transitive inference and reversibility. They rely heavily on specific values in order to reason about, trace and write program code. Many of our think aloud students have exhibited behaviour which is consistent with this type of preoperational reasoning.

We have evidence of students exhibiting characteristics of each of the neo-Piagetian stages in order from least to most mature (Teague & Lister, 2014b). However, our data supports the overlapping waves model which explains why students can exhibit characteristics from two or more stages as they develop skills in the domain (Boom, 2004; Feldman, 2004; Siegler, 1996). In this overlapping waves model, characteristics of the early stage dominate behaviours initially, but as cognitive progress is made there is an increase in use of the next more mature level of reasoning and a decrease in the less mature. In this way, there is concurrent use of multiple stages of reasoning.

One of our series of think aloud sessions used two exercises that required very similar programming skills and we discovered that students who manifested preoperational behaviour were able to complete one, but not the other (Teague & Lister, 2014c). This was because the second task, although functionally equivalent, required the ability to reason about concepts that only someone at the concrete operational level was likely to be able to do.

In-class test data supports our hypothesis that preoperational reasoning may be the norm for novice programmers rather than being peculiar to the small number of students in our think aloud studies. We observed one particular student for several semesters in think aloud studies using a wide variety of programming tasks and were able to find evidence that programming ability improved with the increased ability to reason abstractly about programming code. Our evidence also suggests that it may take several semesters or even years of exposure to programming to develop operational reasoning in this domain (Teague & Lister, 2014b, 2014c). Yet it is at this concrete operational level that we often assume our students are capable of working, and as a result students may be struggling due to inappropriate teaching resources rather than an inability to learn to program.

7. References

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

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program *Computer Science Education Research*. London, UK: Taylor & Francis.

Corney, M., Lister, R., & Teague, D. (2011). *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth. http://crpit.com/confpapers/CRPITV114Corney.pdf

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Paper presented at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia. http://crpit.com/confpapers/CRPITV123Corney.pdf

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

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

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

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

Keats, J., Collis, K., & Halford, G. (1978). Operational Thinking in Elementary Mathematics *Cognitive Development: Research Based on a Neo-Piagetian Approach* (pp. 221-248). Chichester: John Wiley & Sons.

Krathwohl, D. R. (2002). A Revision of Bloom's Taxonomy: An Overview. *Theory into Practice*, *41*(4), 212-218.

Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth, WA. http://crpit.com/confpapers/CRPITV114Lister.pdf

LiveScribe. (2014). Retrieved March 17, 2014, from https://www.smartpen.com.au/

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

Ojose, B. (2008). Applying Piaget's Theory of Cognitive Development to Mathematics Instruction. *The Mathematics Educator, 18*(1), 26-30.

Pea, R. D. (1986). Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research, 2*(1), 25-36.

Piaget, J. (1952). *The Origins of Intelligence in Children*. New York: International University Press.

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

Siegler, R. S., & Crowley, K. (1991). The Microgenetic Method: A Direct Means for Studying Cognitive Development. *American Psychologist, 46*(6), 606 - 620.

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2012). *Swapping as the "Hello World" of Relational Reasoning: Replications, Reflections and Extensions*. Paper presented at the Australasian Computing Education Conference (ACE 2012), Melbourne. http://crpit.com/confpapers/CRPITV123Teague.pdf

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013). *A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers*. Paper presented at the 15th Australasian Computing Education Conference (ACE 2013), Adelaide, Australia. http://crpit.com/confpapers/CRPITV136Teague.pdf

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A., & Lister, R. (2012). *Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming*. Paper presented at the Australasian Association for Engineering Education Conference (AAEE 2012), Melbourne. http://eprints.qut.edu.au/55828/

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

Teague, D., & Lister, R. (2014b). *Longitudinal Think Aloud Study of a Novice Programmer*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand. http://crpit.com/confpapers/CRPITV148Teague.pdf

Teague, D., & Lister, R. (2014c). *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand. http://eprints.qut.edu.au/67314/

Teague, D., & Lister, R. (2014d). *Programming: Reading, Writing and Reversing*. Paper presented at the ITiCSE '14, Uppsala, Sweden.