On the Cognitive Development of the Novice Programmer

and the Development of a Computing Education Researcher

Raymond Lister
School of Computer Science
University of Technology, Sydney
Sydney NSW Australia
Raymond.Lister@gmail.com

ABSTRACT

This paper is a companion to my keynote address at the 9th Computer Science Education Research Conference (CSERC '20). I review the research that led to my three stage neo-Piagetian model of how novices understand code. Code tracing is the key. In the first stage, the novice cannot trace code. In the second stage, the novice has mastered tracing, but, crucially, that is the only skill they have mastered. It is only when novices reach the third stage that they begin to reason about code in a more general, abstract way. The principal failure of traditional approaches to teaching programming has been the assumption that the novices begin at the third stage.

CCS CONCEPTS

Social and professional topics → Computing education

KEYWORDS

Neo-Piagetian theory, novice programmers.

ACM Reference format:

Raymond Lister. 2020. On the Cognitive Development of the Novice Programmer: and the Development of a Computing Education Researcher. In *Proceedings of the 9th Computer Science Education Research Conference (CSERC '20). ACM, New York, NY, USA, 15 pages.*

1 Introduction

The poet Thomas Gray wrote "*Where ignorance is bliss, 'tis folly to be wise*". For my first four years of university teaching I was indeed blissfully ignorant. My class survey results were good. I was nominated for a university teaching award. I believed I taught well.

Three factors conspired to ruin my bliss. The first factor was my persistent high failure rates. If my lectures were the very model of clarity, then why were a third of my students failing? The second factor was the criticism of my colleagues. As surely as farmers complain about the weather, computing academics will complain about the brokenness of the introductory programming subject. I recall many academic staff meetings to discuss the parlous programming ability of our students. Each year everyone voiced their intuition on how students could better learn programming. Year after year, each academic repeated the same intuitions. Every year we made changes, but we were not making progress. Rather than moving forward, we were conducting a random walk in pedagogical space.

The third factor, which absolutely smashed my bliss, occurred in the year 1999. That year, the class was large, and the school's budget was small, so we could not hire the army it took to grade the final exam. With reluctance, we decided that the final exam for the introductory programming class would consist entirely of multiple-choice questions. Since the students would not have to write any code, I worried that the exam would be too easy. To my astonishment, the students did poorly on the multiple-choice exam. There were things going on in my class that I did not understand.

By this time, I had been teaching for five years, so I clearly was not learning what was happening in my class simply from teaching the class. I had to take a step back. If I was to learn what was going on in my class then, like the lead character from the movie, "The Martian", I was going to have to "science the shit out of it". To quote another space figure, but this time a real person, Werner von Braun, "Research is what I'm doing when I don't know what I'm doing". Twenty-one years after that multiple-choice exam, I now say, "Education research is what I'm doing when I don't know what I'm teaching".

2 Bootstrap & the McCracken Working Group

My nascent computing education research career was turbocharged in the year 2002, when I was one of twenty academics accepted into the "Bootstrap" project [3, 8]. Bootstrap was led by Sally Fincher, Marian Petre and Josh Tenenberg. Their aim was to introduce a critical mass of computing academics to education research and thus bootstrap a new computing education research community, to recover from the computing education research "winter" of the 1990s. Our two workshops were held in a boy scout hut, in the beautiful village of Port Townsend, Washington, USA.

Among the readings for Bootstrap was a paper written by an ITiCSE 2001 working group, led by Mike McCracken [23]. The McCracken group assessed the programming ability of a large population of students from several universities, in the United States and other countries. The authors had students write code on a common set of programming problems. They found that most students performed much more poorly than expected. Most students did not even get close to a complete, correct solution.

The McCracken paper was a game changing paper for me. Prior to the McCracken paper, if you were brave enough to say that your students could not program, the resulting conversation usually revolved around what it was that you were doing wrong; how you, an individual, might fix what was your problem. But when students are pooled from across institutions and countries, you begin to discern fundamental patterns.

Although I was inspired by the McCracken working group, I did not agree with its conclusion. The McCracken group attributed the poor performance of most students to an inability to problem-solve. That is, an inability to carry out a five-step process: (1) Abstract the problem from its description, (2) Generate sub-problems, (3) Transform sub-problems into sub-solutions, (4) Re-compose, and (5) Evaluate and iterate. My own experience with a multiple-choice exam led me to wonder whether some of my students lacked abilities that were a precursor to problem-solving. But I could not be sure. The results I saw in my students might be due to poor teaching by me. After talking to several of my fellow bootstrappers, who had similar teaching experiences to me, we decided to convene our own ITiCSE Working group, to investigate the issue further.

3 THE LEEDS WORKING GROUP

Our Leeds Working Group collected data from over 600 introductory programming students, spread across 12 institutions in 7 countries [14]. The students were asked to answer several questions that were placed into their end-of-first-semester exam. The questions were from my multiple-choice exam.

Not all the multiple-choice questions required students to trace code (i.e., manually execute code "on paper"), but the results from the tracing questions turned out to be the most interesting. The working group found that most students at all the participating institutions could not trace reliably. Prior to this multi-national study, it was taken for granted around the world that most students could do the "simple" things, like trace code. Applying Occam's Razor, and with the confidence founded in data from 12 institutions in 7 countries, the working group argued that McCracken's attribution to students of a weakness in problem solving was an unnecessarily sophisticated explanation. The simpler explanation was that such students lacked abilities that were a precursor to problem-solving, at least the skill of tracing code.

3.1 Doodles and Think Alouds

Earlier, in the boy scout hut in Port Townsend, when we Bootstrappers had first talked about convening a working group, I thought my disagreement with the McCracken paper was methodological – I thought they should have screened their students with a pre-test and then eliminated students who could not trace code. I thought of the screening process as eliminating the students who had not studied hard enough. But when we analyzed all three forms of our data in Leeds, I began to wonder whether the issue was deeper than a disagreement over method. Apart from the "performance" data on whether students answered the questions correctly, we analyzed two other forms of data. One of those other forms was the annotations ("doodles") the students made on their exam papers as they attempted the questions. Many students did not doodle at all. When I trace code, I doodle to track changing variable values – why were students not also doodling?

Another form of data the Leeds Group analyzed were transcripts from think aloud sessions with some of the students. When we included the think alouds in the study design, my interest was in studying the metaphors used by students to describe code. I expected statements like "the loop index fell off the end of the array". To my surprise, students did not use metaphors – none! By whatever method students understood code, they were not thinking about code the way I thought they did.

3.2 Reflections on the Leeds Working Group

The Leeds Working Group is well known for its results from the performance data, but I do not think the doodle and think aloud data ever received the attention from readers that was warranted.

My interest in studying metaphor, only to find no metaphor in the transcripts, highlights that I was out of touch with my students at the time of the Leeds Working Group. Back then, I had the same blind spot that many computing academics have – I thought I knew my students. Clearly, I did not. No teacher of programming learns everything they need to know about their students exclusively from teaching those students, no matter how many years of experience the teacher has accumulated. At the time of the Leeds Group, I had accumulated 10 years of teaching, but it may have been more accurate to say that I had taught for 2 years, 5 times.

A common criticism of the Leeds Working Group has been that several of the questions involved code that students would not see "in the real world". While I do not agree with that criticism, I accept that it is a reasonable opinion. It was disappointing, however, that none of the people who made that criticism did not repeat the Leeds Group study, using questions they felt were more appropriate. While no individual who made such a criticism is obliged to do such a study, that nobody in the whole computing education research community (to my knowledge) who made that criticism felt a need to do such a study was an indication that the community, as a whole, was not (at least back then) grounded in a tradition of evidence-based discourse. Back then, if not now, I think that researchers in the computing education community tend to accept or reject the research of others based on whether the results were consistent with their teaching intuitions. Someone reading this paper might think to themselves that my initial disagreement with the McCracken paper was intuitive. That is correct, but the Leeds Working Group went on to do research that turned intuition into something solid. Of course, no individual can do research to rebut everything that is contrary to their intuition. My disappointment, at least back then, if not now, was that nobody went beyond intuition to do that sort of follow-up research.

Over the years, people have raised the question with me whether it is possible to compare students across institutions. I invite such people to look at Figure 3.6 in the working group's paper [14]. That figure is reproduced in this paper as Figure 1. It shows the percentage of students who answered each multiple-choice question correctly for six of the participating institutions. The data from one of those six institutions differs markedly from the other five institutions (i.e., the institution with an exceptionally low

percentage of students answered question 4 correctly). For the other five institutions, beginning at question 4, the lines connecting the data points of each institution show a common pattern – the lines go up slightly from question 4 to question 5, then down to question 6, then up to question 7, down to question 8, and up to question 9. Thus, while the absolute percentage of students who answer a question correctly varies considerably between institutions, the relative performance between questions show the same general trends. This observation from Figure 1 leads me to the following conjecture:

> Consider two institutions, where the students are given the same two exam questions. Suppose there is a statistically significant difference in the performance of students on those two questions within each institution. Then whichever was the harder question at one of the institutions will also be the harder question at the other institution.

Some commonsense qualifiers must be applied to the conjecture. For example, the conjecture should not be expected to hold if students at one of the institutions had been shown code included in one of the questions prior to the exam, while the students at the other institution had not seen that code. Another example would be if the teaching at one institution emphasized object-oriented concepts while the teaching at the other institution emphasized procedural concepts. If the reader is not comfortable with the arbitrariness of applying common sense, then I am content with turning the conjecture around: if the conjecture holds between two institutions, then the content and method of instruction at both institutions are broadly the same, at least for the content tested by the two questions.

In recent years, some in the community have argued that all empirical research papers should characterize the type of institution(s) at which data was collected from students (e.g., "a small teaching focused liberal arts college", or "a large state-funded, research focused university"). I disagree. Even when the type of institution may be a factor, I believe screening tests should be used to eliminate institutional bias.

As a researcher, the year 2004 was a good year for me. Ironically, however, it was a bad year for me as a teacher. Citing failure rates in my class that he regarded as too high (between 30% and 40%), the head of my school moved me off teaching introductory programming. I would not teach another programming class for ten years, until that head of school retired.

4 BRACElet (1): Explain in Plain English

Shortly after the Leeds Group, Tony Clear invited me to present the group's work at a two-day workshop he hosted at the Auckland University of Technology. The workshop was held in December 2004. From that workshop, the BRACElet* project emerged, led by Tony Clear, Jacqueline Whalley and me [5, 18, 38].

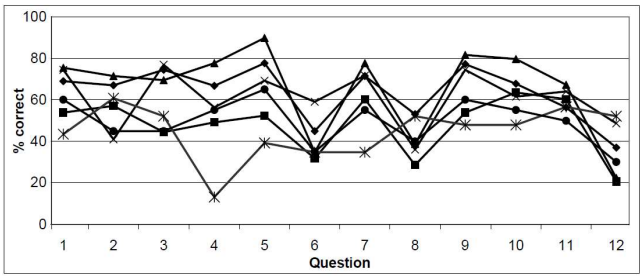

Figure 1. A graph from the Leeds Working Group. The percentage of students with the correct answer for each question, for the 6 institutions that provided data for at least 20 students. Each trend line corresponds to one institution.

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

```
bool bValid = true;

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

Figure 2. An "explain in plain English" question

BRACElet set out to answer the question that the Leeds Working Group (implicitly) begged to be asked – *apart from tracing, are there other precursor skills to code writing?* To address that question, BRACElet introduced a new type of question, the "explain in plain English" question. At the first BRACElet workshop, I remember we discussed for hours how we might probe at a student's ability to read and understand code. After many complex suggestions had been floated, only to be shot down, and with homeward plane flights looming, someone suggested that we simply show the students some code and ask them to explain what the code does "in plain English". We then tossed around ideas of what piece of code we should ask the students to explain. Someone suggested the code shown in Figure 2, for which a suitable explanation would be "it checks to see if the array is sorted". The suggestion of that code turned out to be a wonderful choice. We were partly lucky, I suppose, in making that choice, but I suspect that we were also expertly guided by our collective experience as teachers.

4.1 The SOLO Taxonomy

BRACElet participants wanted a principled way of analyzing the answers to the question in Figure 2. At the second BRACElet

* The BRACElet project is often confused with the BRACE project. Given the similarity in the names, that is understandable, but in fact there is no overlap between the research conducted in the BRACE and BRACElet projects. BRACE was a rerun in Australia and New Zealand of the "Bootstrap" and "Scaffolding" projects [3, 8]. BRACE stands for "Building Research in Australasian Computing Education". BRACElet was initially intended as a smaller follow-up to BRACE, for academics in New Zealand who had not been able to attend the two BRACE workshops in 2004 and 2005 — hence the "let" part of BRACElet. However, BRACElet took on a separate life of its own. Between 2006 and 2010, BRACElet produced 16 papers with a total of 26 different authors, from 14 different institutions, across 7 countries.

workshop, six months later, we settled on using the SOLO taxonomy [2] to analyze the student responses to the "explain in plain English" question. The SOLO taxonomy categorizes student answers into five levels which are, from least to most sophisticated:

1. **Prestructural:** The response by the student could have been provide by someone who had not studied the subject.
2. **Unistructural:** The student manifests a correct grasp of some small part of the problem.
3. **Multistructural:** The student manifests an understanding of most parts of the problem but does not manifest an awareness of the relationships between these parts. The student fails to see the forest for the trees. A line-by-line description of the code in Figure 2 is a multistructural response.
4. **Relational:** The student integrates the parts of the problem into a coherent structure and uses that structure to solve the task. The student sees the forest. For the code in Figure 2, the answer "it checks to see if the array is sorted" is relational.
5. **Extended Abstract:** The student goes beyond the immediate problem to be solved and links the problem to a broader context. This category was not studied by BRACElet.

4.2 The Prerequisites for Code Writing

The first two papers published by BRACElet [37, 15] describe the results and conclusions from this first round of work by BRACElet. One of the results was that students who gave a relational response to the explain-in-plain-English question in Figure 2 tended to perform better on the exam as a whole. In the conclusion of the second BRACElet paper [15], we speculated:

> In our view, students who cannot read a short piece of code and describe it in relational terms are not intellectually well equipped to write similar code.

Thus, in this opening phase of BRACElet, we speculated that there was a linear hierarchy, where the ability to trace code reliably preceded the ability to explain code reliably, which in turn preceded the ability to write code reliably. In the next experimental phase of BRACElet, we would see results that supported a hierarchy, but we would find out we were wrong about it being a linear hierarchy.

5 BRACElet (2): TRACING AND EXPLAINING

To empirically study the relationship between tracing, explaining and writing code, BRACElet constructed a new set of exam questions that included all three types of questions. The first substantive discussion of exam data that included all three types of questions was at the sixth BRACElet workshop, in December 2007. I remember that a preliminary analysis of exam data was presented on day 1 of the workshop, but the breakthrough was presented on the second morning. Overnight, Mike Lopez had put the data through a statistical analysis tool he had written, which performed an analysis similar to structural equation modelling. Mike found that code tracing questions alone did not correlate well with student scores on code writing, nor did explain-in-plain-English questions alone correlate well with student scores on code writing. However, a linear combination of student scores on code tracing and code explaining did correlate well with code writing [21].

Mike's way of analyzing the data was too sophisticated for me to repeat, but after he had shown us what to look for, a linear combination of scores on tracing and explaining correlating with code writing, simpler methods could be used to look for that in other data. By mid-2009, we had published papers reporting the same relationship in two more datasets [16, 35].

At ITiCSE 2009, we convened a BRACElet working group [17]. About half of the twelve working group members had not participated in BRACElet before, and those people brought fresh ways of thinking to BRACElet. For me, the most stimulating aspect of that fresh thinking was a review of education theories that might be applicable in understanding the development of the novice programmer, theories originating from other disciplines, especially mathematics. That discussion of theory by the working group planted a seed in my thinking that would grow to dominate my research for the next five years.

6. A THREE STAGE MODEL (EXPURGATED)

Stimulated by the ITiCSE 2009 working group, I began to think about theories that might explain the empirical results from BRACElet. Intrinsic to SOLO is the principle that students can acquire facts without (at least initially) integrating those facts with the rest of their knowledge. Beyond that, however, SOLO does not tell me any more about how novice programmers think and learn. I felt I needed to move beyond the SOLO taxonomy.

In mid-2009, I had taken a year's leave without pay from UTS to try a different job. By the end of 2009, the new job had not worked out, and I found myself "on sabbatical" (i.e., under employed) for six months until I could return to UTS mid-2010. While I did not know it at the time, this six-month sabbatical was fortunate, as it gave me the chance to read from the large piles of papers and books that I had collected but never had time to read.

Eventually my reading led me to construct a three-stage model of the mental development of novice programmers, which I published in 2011 [19]. I then went on to look for evidence for my three-stage model, in conjunction with a Ph.D. candidate, Donna Teague [32], now Donna Kingsbury. I describe the three stages in the following subsections.

6.1 Stage 1: Pre-Tracing

In the initial stage, pre-tracing, the student has a sparse and incoherent understanding of programs. At this stage, students exhibit a haphazard approach to writing code, and cannot reliably trace code, for several reasons:

- *Misconceptions of how programs work*. For example, a novice might think that the assignment statement "`x = y`" links those two variables so that any subsequent update to one variable also updates the other variable. Many common misconceptions are documented in the existing literature. See appendix A of Juha Sorva's thesis for a catalogue of over one hundred misconceptions [31].

- *Anthropomorphizing the computer*. A novice programmer can behave as if an intelligent entity lurks inside the computer; an entity that somehow knows what the novice wanted their program to do. Such thinking by a novice programmer is natural in a world where word processing software corrects our spelling and search engines make suggestions about what search term we really meant to type. Pea [26] referred to this anthropomorphizing of the computer as a type of "superbug", a bug that transcends any program or programming language.
- *Programming as* (*Witch*) *Craft.* Programming is often described by experts as being a craft. For the pre-tracing student, programming is witchcraft. Wikipedia describes "Voodoo programming" as being the "practice of getting a program to produce desired output by using guesses, trial-and-error, cookbooks, copy-pasting from online resources, or similar techniques without truly understanding the underlying problem". Wikipedia describes "Cargo cult programming" as being a "style of computer programming characterized by the ritual inclusion of code or program structures that serve no real purpose". In the case of cargo cult programming, I have often had the experience of asking a student to explain to me the purpose of a strange line of code in their program, only to be told "I don't know what it does, but if I leave it out the program doesn't work".

6.2 Stage 2: Tracing

By the second stage, tracing, the student can reliably trace code. However, the stage 2 student tends not to abstract from the code itself. For the stage 2 student, there is nothing but the code. The only way that a stage 2 programmer can reason about a piece of code is by tracing that code. Over-reliance on a single way of thinking is known in psychology by several names, including "Maslow's Hammer". Maslow probably appropriated an old proverb when he wrote:

> I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail. [22, p. 15].

Because all that stage 2 programmers can do is trace code, they reason by induction. That is, when attempting to explain what a piece of code does, the stage 2 programmer (1) generates a set of initial variable values, (2) traces the code, and then (3) attempts to infer the function of the code by comparing the initial and final values. For example, in the exercise shown in Figure 3, the variables names "`y1`", "`y2`" and "`y3`" will often lead the stage 2 novice to perform a trace using the initial values `y1=1`, `y2=2` and `y3=3`. Tracing with those initial values results in final values `y1=3`, `y2=2` and `y3=1`. Teague found that when stage 2 programmers used those initial values, they often answered incorrectly that the code reverses the order of the values in the three variables [32].

A novice programmer who traces the code in Figure 3 only once and arrives at the wrong answer is an early stage 2 programmer. Early in stage 2, the effort of tracing code is great and so the novice tends to only perform a single trace. As dexterity at tracing improves, the stage 2 novice becomes willing to perform more than a single trace. Teague [32] reported on a think aloud session with

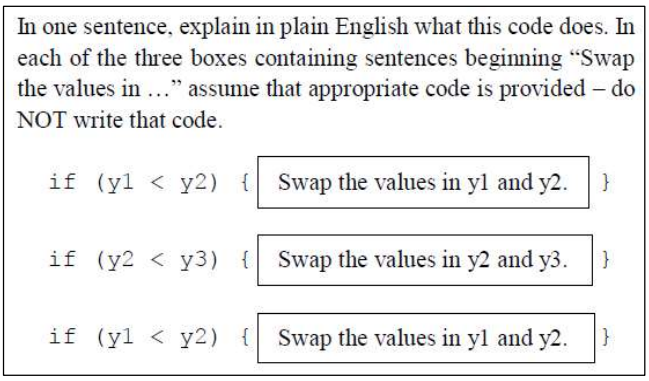

Figure 3: Code for sorting three integer values.

two students "Lucas" and "Sierra", who worked together on the problem in Figure 3. Lucas and Sierra began by performing a trace with the initial values `y1=1`, `y2=2` and `y3=3`. After that trace, Sierra jumped to the wrong conclusion. Lucas, however, insisted on performing a second trace and then arrived at the correct answer.

Stage 2 novices use that same inductive approach when attempting to debug their own code. That is, stage 2 novices trace their buggy code with specific values, and then make what is often a myopic patch. That patch may "fix" the code for the specific initial values just used in the trace, but the patch may not address the general bug [9]. The strategy of "repeat-trace-patch-until-success" is like "shotgun debugging", which Wikipedia defines as, "A process of making relatively un-directed changes to software in the hope that a bug will be perturbed out of existence".

6.3 Stage 3: Post-Tracing

In the third stage, post-tracing, novices begin to reason about code the way we have always assumed they did – deductively, from reading the code, and/or from simple diagrammatic representations of operations on data structures. This stage is the first stage where students show a purposeful approach to writing code. Therefore, the stage 3 programmer may further develop their programming skills using the approach used for decades in universities — having them learn by writing lots and lots of code.

6.4 A Reflection on the Expurgated Model

At this point, the reader might be thinking that the three-stage model is either trivial or arbitrary. If the reader thinks so, that is because this "expurgated" version of the model has been presented in the absence of background theory that both justifies the structure and fleshes out further detail. The next section of the paper presents the model again, but with the theoretical background. The reader may ask: why do I present my model twice, first without the underlying theory and then with theory? The answer is hinted at in the title of this section, by my use of the word "expurgated". Just as a novel might be expurgated to remove words that may cause offense, when I introduce someone to my three-stage model, I have learned to do so initially by omitting the "P" word, which sometimes causes offense – "Piaget".

7. A THREE STAGE MODEL (PIAGETIAN)

During my "sabbatical" in 2010, I was led to the type of theory I was looking for when I read deeper into the SOLO taxonomy. In an appendix of their book [2], Biggs and Collis describe how they had derived SOLO by eliminating the structuralism of Piagetian theory. That appendix led me to read Piaget, and then read neo-Piagetian theory, from which my stage model emerged [19, 20].

When presenting my model, I have found that invoking Piaget often meets with audience resistance. Piaget is out of fashion. There are a few reasons for this, but the principal reason is that Jean Piaget worked in the early-to-mid 20th century intellectual period when structuralism was dominant, and so his direct legacy is not well received in our current period of post-structuralism. Another reason is that much of Piaget's writings about babies has been refuted by empirical research in recent decades, which has (unfairly) decreased the perceived credibility of Piaget's work in general. Piaget's fall from fashion is ironic, as most computing education researchers emphatically describe themselves as being constructivists, yet Piaget is the father of constructivism. (At the time of writing this paper, googling "who is the father of constructivism" is answered with "Piaget" – QED.)

Since Piaget's death, the neo-Piagetians have further developed Piaget's work, improving the compatibility with both post-structuralism and with observational data [25]. Jean Piaget's "classical" theory and neo-Piagetian theory both describe cognitive development in terms of sequential, cumulative stages, but neo-Piagetian theory differs in several ways, summarized in Table 1.

7.1 Stage 1: Sensorimotor (Pre-Tracing)

Piaget was interested in the intellectual development of children. Anyone who has watched a baby stare at an object, before painstakingly reaching out to grasp that object will be comfortable with Piaget's use of the term "sensorimotor" in that context. I have found, however, that many people are not comfortable with how "sensorimotor" applies to people of any age who have just begun to learn to program. A glib answer is that, while these novice programmers may be adults in physical space, they are babies in cyberspace. A more serious answer follows. This first Piagetian stage of learning to program is a struggle to learn to correctly perceive and trace code. The "sensori" component of the name is appropriate because novice programmers do not initially perceive code the way that an accomplished programmer does. For example, novices at this stage often read code as static text, not as executable code.

The "motor" component of the name is appropriate because these sensorimotor novice programmers do not, for example, trace code the way that an accomplished programmer does. Teague discovered that when novices do perform a trace with pen and paper, they can use *ad hoc*, error prone ways of recording the changing variable values [32].

While there is extensive literature on the misconceptions of novice programmers [31, see appendix A] that literature under-represents some issues about novice programmers that come into focus with a Piagetian perspective. One of these issues is the low level of commitment that a sensorimotor programmer has to their conceptions about programming, whether those conceptions be right or wrong. Instead, the sensorimotor programmer can swap between (mis–) conceptions, based on superficial aspects of the code. Furthermore, the sensorimotor programmer does not merely have misconceptions – the sensorimotor programmer can have a different way of conceiving code. For example, as mentioned before, many sensorimotor programmers read code as static text. In Piagetian terms, this non-dynamic reading of code is known as "figurative intelligence".

Brooks proposed a theory of how accomplished programmers comprehend unfamiliar code [4]. According to Brooks, one strategy used by accomplished programmers is a search for "beacons", which are data structures or operations that verify a hypothesis about the code. For example, a loop in which values in an array are swapped around is a beacon for sorting. In contrast, the sensorimotor novice does not make informed hypotheses about code. Such a novice does note features in the code, but such a feature is not really a beacon; more a talisman (i.e., an object possessing mystical power). Traynor, Bergin, and Gibson [34] provided a telling quote from such a student, who described how he/she went about answering exam questions that required coding:

> … 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".

7.2 Stage 2: Pre-operational (Tracing)

To understand what "operational" means in the Piagetian stage name "pre-operational", consider a machine being installed in a factory: the machine is not yet "operational". Likewise, at the pre-operational stage the novice programmer is not well placed to independently write code of their own. Note, however, that this does not imply that such a novice should not write code as part of the learning process – on the contrary, a novice must write code to learn how to code. However, we should expect that pre-operational students will write code that is either trivial or code that is poorly designed and buggy.

The fresh insight that comes from neo-Piagetian theory is the identification of the pre-operational programmer as a natural stage of progression for a novice programmer, not an anomalous behaviour. Furthermore, students can spend a long time in the pre-operational stage. Novices may only progress beyond the pre-operational stage after tracing through many examples of code.

In the absence of neo-Piagetian theory, it is very difficult for teachers to understand why pre-operational students cannot use diagrams to understand code. For example, Thomas, Ratcliffe, and Thomasson wrote despairingly of their frustrations at trying to get their novices to make effective use of diagrams:

> Providing ... 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'. [33, p. 253]

Classical Piagetian Theory by Jean Piaget	Neo-Piagetian Theory
Is concerned with the general cognitive development of children.	Is concerned with the cognitive development of people of any age as they learn any new cognitive task.
A child at a particular Piagetian stage applies the same type of reasoning to all cognitive tasks (e.g., math and chess), apart from exceptions known as *décalage*.	Since a person's cognitive ability in any domain is a function of their degree of learning in that domain, a person will exhibit different Piagetian stages in different knowledge domains.
There are typical age ranges for each Piagetian stage, but empirical evidence shows great variation in age ranges.	Age ranges are not prescribed. But there may be minimum ages before which manifesting a particular stage in any domain may be considered exceptional.
Children spend an extended period in one stage, before undergoing a rapid change to the next stage – the "staircase" metaphor.	Over a short period of time, people may exhibit a mix of stages. As learning progresses, the frequency of manifestation of higher stages will increase − the "overlapping wave" metaphor.

Table 1. A Comparison of Classical and Neo-Piagetian Theory.

For the reader who would like to learn more about the pre-operational programmer, see the collection of papers in Donna Teague's thesis-by-publication [32].

7.3 Stage 3: Concrete Operational (Post-Tracing)

As "operational" in "concrete operational" implies, when a novice reaches the third stage, they can independently write reasonably well-designed code. It is only at this stage that the novice programmer begins to reason about abstractions of code, such as diagrams.

Recall that, as pre-operational programmers develop, they are more likely to perform multiple traces on a piece of code, with different initial values intended to exercise different pathways through the code. The transition from pre-operational to concrete operational may begin with the novice consciously choosing initial values to represent a class of possible initial values. For example, in some hypothetical piece of code, the novice might choose to initialize two variables "`a`" and "`b`" as `a=1` and `b=2` to represent all possible initial values where `a < b`.

A novice has made the transition to the concrete operational stage if, while reading through a piece of code, the novice no longer uses specific initial values, but instead mentally maintains informal but algebraic-like constraints on the possible values in each variable. For example, consider a student studying the three "`if`" statements in Figure 3. After the first of those if statements, the concrete operational student would think of `y2` as holding any possible value that satisfies the condition that it is less than the value in `y1`. Teague [32] refers to this as "abstract tracing".

Earlier, when I introduced the explain-in-plain-English question shown in Figure 2, I wrote that we were lucky to have made that choice, but we were also guided well by our collective experience as teachers. Piaget offers an explanation for why the code in Figure 2 proved to be a good choice. If the variable `bValid` is to maintain its initial value, then `iNumbers[0]` ≤ `iNumbers[1]` ≤ `iNumbers[2]`… and so on. By the rule of logic known as transitive inference, those inequalities imply that the elements of the array are sorted. Piaget maintained that the ability to perform transitive inference is one of the defining qualities of a person at the concrete operational stage.

7.4 Overlapping Waves

According to neo-Piagetian thinking, novice programmers should not be classified as being at a unique stage of development at any given moment (i.e., sensorimotor, pre-operational, concrete operational). Instead, neo-Piagetians advocate an "overlapping waves" model [29], where a person exhibits an evolving mix of the Piagetian stages. The concept of overlapping waves is illustrated in Figure 4. When a person begins their study of a new knowledge domain, they first reason predominantly at the sensorimotor stage, but they evolve to reason less at that stage and more at the pre-operational stage, and so on to later stages. Thus, multiple ways of reasoning coexist.

When the concept of overlapping waves is expressed as a generalization, as it was in the preceding paragraph, it may seem that it renders meaningless the three-stage model. In practice, however, this is not the case. When I have a brief encounter with a student, where I can see quickly what their problem is, and I ask them to trace a portion of their code with specific values, it is usually clear to me which stage is most pronounced in their reasoning at that moment. Some students cannot trace their code (sensorimotor). Others can trace their code but cannot identify the general problem revealed by the trace, or cannot nominate a fix to the code, or they make an inappropriate change to their code (pre-operational). Others either complete the trace or have a "eureka" moment during the trace and do not need to complete it [6], but either way after ceasing to trace they move deliberately to making a plausible change (concrete operational). It is over longer periods of time that a mix of multiple ways of reasoning are usually manifested. It is over even longer periods of time that the mix is seen to change.

In this paper, references are made to novices being in a specific stage. In such cases, the reader should understand that a stereotype is being invoked, for clarity and conciseness. Consistent with the overlapping waves concept, a real student often exhibits a mix of the stages and does not fit the stereotype for a single stage.

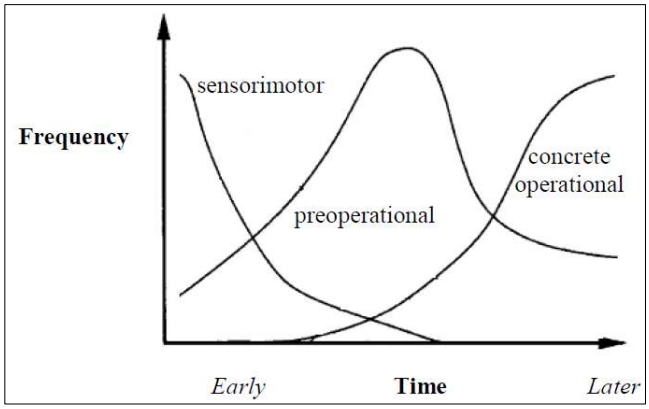

Figure 4. The Concept of Overlapping Waves.

7.5 New Knowledge and Stage Regression

As the novice programmer learns, there are periods of time where the novice may maintain a pre-operational or concrete operational way of reasoning about code even when the novice is taught something new. For example, having learned about integer variables, novices may subsequently learn about floating point variables, without that affecting how they reason about code. In Piagetian terms, this is known as assimilation.

But new knowledge sometimes shatters old ways of thinking. For example, a novice may have a way of recording on paper the tracing of sequential code (i.e., pre-operational stage) but that way of tracing fails when the student is introduced to loops. The novice must now change their way of reasoning about code. As part of the change process, the novice may regress to an earlier stage. The novice programmer must first understand how loops work (i.e., go back to sensorimotor), then devise a new way of recording a trace on paper (i.e., move back up to pre-operational). In Piagetian terms, this regression and recovery is known as accommodation.

7.6 Reflections on the Piagetian Model

A simple but useful summary for an early understanding of how novice programmers think in each of the above three Piagetian stages is as follows:

1. **Sensorimotor:** The code is perceived as static.
2. **Pre-operational:** The code is conceived as changing variable values.
3. **Concrete operational:** The code is perceived as embodying abstract properties that can remain invariant under execution.

The following example is perhaps the simplest possible illustration of the above summary: consider the case where the following three lines of code are given to students, and they are told that the code swaps the values in variables "b" and "c":

```
a = b;
b = c;
c = a;
```

The sensorimotor novice may focus on the fact that the variable on the right of the first and second lines of code are repeated on the left in the next line of code, without understanding why. The pre-operational novice can trace how specific values in the variables change. The concrete operational novice sees that the variable "a" acts as a temporary storage location.

There is a fourth stage, the Formal Operational Stage. This is the ultimate "expert" stage of Piagetian reasoning. This stage is probably more applicable to programmers who are more advanced than students in their first semester of learning to program. As the fourth stage is not to be expected in introductory programming students, it is not discussed in this paper.

There has been some interesting, independent, empirical work that appears to confirm the existence of the sensorimotor, pre-operational and concrete operational stages [27].

There is an old joke about how to cook a chicken, which I shall summarize thus: regularly poor whiskey over the cooking chicken and when the chicken is done, throw the chicken away and drink the gravy. Likewise, some readers might accept the "expurgated" version of the three-stage model but throw away my Piagetian interpretation. Some readers might suspect that neo-Piagetian theory was merely the mental scaffolding that led me to a worthwhile model, but that neo-Piagetian scaffolding may be dispensed with now that the construction of the model is complete. If such thinking leads to the reader to an initial acceptance of my three-stage model, then I am happy. I hope, however, that with time the reader may come to understand that neo-Piagetian theory fleshes out the full complexity of the expurgated skeleton. Prior to reading neo-Piagetian theory, I was baffled by some of my encounters with students; baffled by students who could not explain their own code to me; baffled by students who could not understand a simple diagram I drew for them. I still have the same type of encounters, but now I understand that those students are simply not at the concrete operational stage. Not only am I no longer baffled, but I can now help those students, often by leading them through a trace of their code. For me, the single biggest contribution of neo-Piagetian theory to my teaching practice has been the recognition of the existence of the pre-operational programmer and the lengthy time that some students remain pre-operational.

I sometimes encounter educators who resist the cognitive constructivist ideas of Piaget because they are enthusiastic about the social constructivist ideas of Vygotsky [36]. Both Piaget and Vygotsky are constructivists, and nobody needs to choose between them. Vygotsky placed greater emphasis than Piaget on the role of language and culture in cognitive development, but that emphasis is not a rejection of Piaget. The late Piagetian scholar, Les Smith, sometimes provided entertainment at conference dinners. He would read out several excerpts from the writings of either Piaget or Vygotsky, and the audience members had to nominate which of Piaget or Vygotsky wrote it. Many guessed wrong. Something similar can be said about more recent constructivist theories of education: contemporary constructivist theories are not fundamentally at odds with Piaget or Vygotsky.

8. GENERAL RESEARCH REFLECTIONS

8.1 Writing from Tracing and Explaining

With the help of neo-Piagetian theory, especially the concept of overlapping waves, I believe I now understand how the novice programmer progresses from being primarily pre-operational to primarily concrete operational. The ability to answer explain-in-plain-English questions is a proxy; an estimate of a novice's ability to reason about code in an abstract way. Some people describe the process of acquiring this abstract reasoning skill as a process of acquiring programming plans, often called "schemas". See chapter 4 of Sorva's thesis for a review of schemas in a programming context [31]. As Sorva expresses it, "An introductory course starts the novice on a long road of schema-building" (page 35). While I do not subscribe to all aspects of schemas as an explanation of human reasoning (more on that below), the schema concept is a useful shorthand in the next paragraph.

A novice programmer who is primarily pre-operational has not yet acquired many programming schemas, so such a novice relies heavily on their tracing skill to reason about code. As the novice learns, the novice can use newly acquired schemas to reason about code. However, the novice still needs to rely on tracing when the novice's existing schemas are not applicable. Furthermore, a newly acquired schema may be incomplete, vague, or even buggy, so tracing helps to overcome the inadequacies of newly acquired schema. In the case of a buggy schema, tracing helps the debugging. Also, tracing can be used to fill in the gaps when two or more schemas are combined. As the novice develops a larger set of precise and bug-free schemas, the novice's reliance on tracing decreases – in terms of overlapping waves, the pre-operational wave falls while the concrete operational wave rises. Eventually, the novice becomes primarily concrete operational, and reasons mostly via schemas.

8.2 Non-Computational Models of Learning

Earlier in this paper, I invoked Maslow's hammer [22] when describing how the pre-operational programmer relies on tracing code. The teachers of novice programmers are not immune from Maslow's hammer – when reasoning about human thinking and learning, computing educators are overly reliant on the computer program as a metaphor, which is often referred to as the "Information Processing Model".

Schemas are a computational metaphor. The concept is useful but limited. Just as a Buddhists asks, "What is the sound of one hand clapping?" I ask, "What is the sound of half a schema programming?". Schemas are a "just so" story, like "How the Elephant Got It's Trunk" [13] (Answer: its nose was stretched by a crocodile). To say that a student accessed a programming schema is no better an explanation than saying that the programmer was inspired by god.

Back in the early to mid-1980s, long before I worked in computing education, I worked in good old fashioned Artificial Intelligence (GOFAI) [10]. That is, in the early to mid-1980s, I worked in symbolic AI (not neural networks). I am especially struck by the similarity between GOFAI approaches to writing automatic planning programs and the 1980s work by Soloway, on explaining how novices write programs, in terms of plans and plan merging [30]. I eventually grew disenchanted with GOFAI and with it the computer metaphor for human thinking, especially after I read a book by the philosopher Hubert Dreyfus [7]. It was Dreyfus who prepared my mind to accept non-computational descriptions of thinking and learning, such as Piaget's description.

I suspect that the non-computational nature of Piaget's theories is part of the reason why I sometimes encounter resistance to my neo-Piagetian three stage model. It is the instinct of computer scientists to expect models of thinking and learning to be computational models.

8.3 The Conscious Decision to do Research

I am fond of the play "Life of Galileo", by Bertolt Brecht. I am especially amused by the scene where Galileo tries to persuade a priest, who subscribes to Aristotle's theories, to look through a telescope and see with his own eyes heavenly wonders that are contrary to Aristotle. The priest declines, as he does not see the point of looking through the telescope. Besides, the priest argues, could not what Galileo claims to see be an illusion caused by the telescope itself? I have seen two productions of the play, and in both productions the audience laughs loudly at the priest. The truth is, however, that we should be laughing at ourselves, for we all have more in common with the priest than with Galileo. It is not natural for humans to think as scientists. We are prone to confirmation bias. That is, we place too much emphasis on events that confirm our existing beliefs and too little emphasis on events that are contrary to our existing beliefs.

For me, a particularly telling indication that it is not natural for humans to think scientifically is that, while academics bring a scientific mentality to bear in their research, few academics bring a scientific mentality to their teaching. As I alluded to in the introduction of this paper, at any meeting of academics where the topic is teaching, you will hear academics articulate unsubstantiated intuitions, and relate unconfirmed, biased anecdotes.

There is a popular belief that the first step to overcoming alcoholism is for the victim to admit that they have a problem. Likewise, the first step to becoming a computing education researcher is to admit that you have not learnt how your students learn just from teaching your students, nor have you learnt how they learn from having once been a student yourself.

Parts of this paper are intended as community announcements, in the same community spirit which sees a canoeist hike back upriver, to hammer a sign into a tree, "Warning: Rapids Ahead". Being a computing education researcher is hard. While you abstain from intuition and anecdote, others will not. At meetings of peers to discuss teaching, you will sometimes be the only sober person in the room. Like the priest in Brecht's play, your colleagues will decline to look through your telescope. Like the priest in Brecht's play, your colleagues will argue that what you claim to see is an illusion caused by the lens through which you look. Being a computing education researcher is hard.

8.4 The Importance of a Research Program

When I look at the published research in computing education, I see many authors who, year-by-year, flit from topic to topic. Now, every computing education researcher is free to do that, and some have done excellent work while doing that, but I think our research community as a whole has too few people who are doggedly working on the same research topic, year-after-year. There is an old saying, that an academic is someone who learns more and more about less and less until they know almost everything about almost nothing. As I reflect on my years of studying novice programmers, I am proud that I now know almost nothing.

In ecology, there is the concept of plant succession. For example, the first plants to occupy bare earth are weeds which, having provided suitable pre-conditions, are succeeded by shrubs, which in turn are succeeded by saplings. Eventually, after generations of successions, there is a mature forest. The research I have described in this paper also exhibits a succession. I think it is fair to say, with admiration and not unkindness, that the McCracken group pioneered the bare earth. Starting with the Leeds Group, and thus far culminating in Donna Teague's thesis, there has been a steady succession that has moved us toward a paradigm, with empirical methods and a neo-Piagetian theoretical framework. I certainly would not say that we have reached the point of being a mature forest, but I do hope that we are at least out of the weeds.

8.5 Developmental Epistemology

Piaget used the term "Genetic Epistemology" to describe his work (or at least his translators used that term), but the contemporary understanding of "genetics" often leads to confusion. The expression "the genesis of knowledge" better expresses to the contemporary reader what Piaget intended. To avoid the confusion from using "genetic", I prefer to use the term "Developmental Epistemology". Developmental Epistemology differs from developmental psychology in that the former emphasizes the knowledge of a specific domain, while the latter emphasizes aspects of learning that transcend knowledge domains.

If we are all constructivists now, then Developmental Epistemology should be a central research area of every discipline. Piaget said, "there exists no structure without a construction". If computing academics aspire to teach a large percentage of the total student population, then we need to understand how those students can efficiently construct our body of knowledge in their own minds.

9. A TEACHING EXAMPLE

In 2014, a new head of school returned me to teaching the introductory programming subject. If the proof of the pudding is in the eating, then the proof of education research is in the teaching, so my return to teaching introductory programming gave me the opportunity to find out if my three-stage model was of any use.

In this section of the paper, I will provide one example of how I now teach iterative processes on arrays in a way more consistent with my three-stage model. Aspects of my teaching were also influenced by a paper by Walter Milner [24].

9.1 Pre-Requisite Knowledge and Skills

Page limits do not allow me to provide every detail, so in the example to follow, I shall assume that students already understand Sequence, Selection and Assignment. I have described aspects of how I teach those topics in an earlier paper [20]. Also, page limits prevent me from providing a description of how I teach methods/functions/procedures, which is a difficult concept for many novice programmers, especially the use of parameters.

9.2 Arrays Early

I introduce arrays in week 2. Here is the first piece of Java code using arrays that I discuss with the students:

```
int [] a = {5, 7};
a[0] = 3;
```

In week 2, I only use array subscripts that are constants. I have found that students have no more difficulty understanding arrays with constant subscripts than they do with understanding scalar variables. In weeks 2 and 3, I illustrate arrays with code such as the following, which shifts the elements of an array one place to the left, with the leftmost element rotating to the rightmost position:

```
temp = a[0];
a[0] = a[1];
a[1] = a[2];
a[2] = temp;
```

9.3 CountElement 1 (constant subscripts)

I now illustrate my method of teaching iterative processes on arrays via a method `countElement`, which counts the number of times a given value occurs in an array called "`list`". The code I show below is Java, but the programming language is not important.

Figure 5 shows the first version of `countElement` presented to students. The code uses constant subscripts. The array "`list`" must have exactly four elements. Figure 5 includes a trace table for students to complete. Students are required to trace this code, sometimes with several sets of given initial values.

Introducing arrays with constant subscripts familiarizes students with basic array concepts before they are introduced to variables as subscripts. When the students eventually move to using variable as subscripts, their prior experience with constant subscripts reduces the danger that students will confuse the position in an array with the value stored at that position.

9.4 CountElement 2 (variables as subscripts)

I then introduce a version of `countElement` that still only sorts arrays of size 4 but uses variables as subscripts. Figure 6 shows the code and trace table given to students. With this new version (and also with subsequent versions) students may regress to the pre-tracing/sensorimotor stage when they first encounter variables as subscripts. After coming to understand variables as subscripts, students rise again to the same stage they attained when working with constant subscripts.

```
/**
 * @return    the number of times the value in "val"
 *            occurs in the four-element array "list"
 *
 * @param     val  contains the value to be counted
 */
public int countElement (int val)
{
  int count = 0
  if (list[0] == val) ++count;
  if (list[1] == val) ++count;
  if (list[2] == val) ++count;
  if (list[3] == val) ++count;
  return count;
}
```

true/ false	count
	0

Figure 5: Version 1 of a method "countElement" on a list containing four elements using constant array subscripts.

```
int i = 0;
int count = 0;

if(list[i] == val) ++count;
++i
if(list[i] == val) ++count;
++i
if(list[i] == val) ++count;
++i
if(list[i] == val) ++count;
++i

return count;
```

true/ false	i	count

Figure 6: Version 2 of a method "countElement" on a list containing four elements using variable "i" as the array subscript.

This second version is longer and more complex than the first version. I justify this version to students in two ways: (1) if we wanted to increase the size of the array on which `countElement` could operate, this second version merely requires us to copy-and-paste some lines, without any edits to the lines added, (2) I simply confess that the real purpose of this version of `countElement` is to make the next version easier to understand.

9.5 CountElement Versions 3 and 4 (loops)

The third version of `countElement` uses a loop. Initially, the loop still only sorts arrays of size 4:

```
for (int i=0 ; i < 4 ; ++i)
   if (list[i] == val) ++count;
```

The above code leads to the final version, which works on an array of any length, by replacing the 4 with `list.length`.

9.6 Thirteen Examples for Listof4 / ListOfN

A common failing in the teaching of programming is that students are required to write original code before they have been shown enough examples from which to generalize. A rule-of-thumb is that students require seven examples before they begin to generalize. (The number 7 here is not to be confused with the 7 plus or minus 2 commonly associated with the capacity of short-term memory.) While the rule-of-thumb is not derived from Piagetian theory, it appears consistent with Piagetian theory. A student reasoning primarily at the concrete operational stage may only need one or two examples. However, a student reasoning primarily at the pre-operational stage will probably need several more examples.

The `countElement` method described above is one of 13 methods I teach as part of introducing students to iterative processes on arrays, with all 13 methods taught the same way as `countElement`:

1. **`copyList`:** copies one list to another
2. **`countElement`:** as described above in this paper
3. **`findUnSorted`:** returns the first position occupied by a given value, or -1 if it is not found
4. **`insertFirst`:** inserts a new value in the first position. The other values are pushed up one place, and the value that was previously last is lost.
5. **`insertLast`:** inserts a new value in the final position. The other values are pushed down one place, and the value that was previously first is lost.
6. **`minVal`:** returns the minimum value
7. **`minPos`:** returns the position of the minimum value
8. **`printList`:** outputs the entire list
9. **`replaceAll`:** replaces all occurrences of a given value with a new value
10. **`replaceOnce`:** replaces the first occurrence of a given value with a new value
11. **`reverseList`:** reverses the order of the values
12. **`sumList`:** returns the sum of the elements of the list
13. **`toString`:** returns all the values as a single String.

9.7 The Keller Plan

For 70 of the 100 points that determine a student's grade, I have created a self-paced "Keller Plan" [12]. For the purposes of this paper, the essential element of a Keller Plan that the reader needs to understand is that the knowledge to be learnt is broken into a sequence of units, and a student must pass a test on each unit before that student is allowed to move on to the next unit. The student may attempt a test for a unit as many times as required.

To earn the 70 points taught via a Keller plan, a student must pass 30 short lab tests under exam conditions. The 13 methods of `ListOf4` comprise 2 of the 30 lab tests. The remaining 30 points are earnt via a conventional code writing assignment, which students may only attempt after completing the 30 lab tests. To pass the subject, a student need only score 50 of the available 70 from the Keller Plan. However, students who intend to go on to further study of programing are advised to complete the full 70 points.

The 30 lab tests are graded, automatically by computer, as either "pass" or "fail" – there is no intermediate result. To achieve a pass, the student's code must pass all test cases put to the student's code by the automatic system.

Since the students may repeat a lab test as many times as they need, the lab tests are not presented as a previously unseen code writing problem. On the contrary, the students are given a model solution to each lab test before they make their first attempt. Many readers will be troubled that the 30 lab tests do not require students to write code of their own devising, with the fear that students who pass the 30 lab tests have merely engaged in rote learning and may not understand the material they have passed. I have three responses to that criticism:

- Empirical data supports my use of a Keller plan. The programming class that follows my programming class uses a conventional approach to teaching programming and has not been altered to accommodate my approach to teaching. In that following class, there is no statistically significant difference in the failure rates between those students who completed all 30 lab tests but who did not attempt the code writing assignment (i.e., achieved exactly 70 points) and those students who did attempt the code writing assignment and scored a total of 85 or higher.
- To believe that having students reproduce code inevitably leads to memorization without understanding is to confuse rote learning with "meaningful learning" [1]. Consider the 13 methods listed above, which comprise 2 of the 30 lab tests. It certainly may be possible to memorize the 6-7 methods that comprise each of those lab tests, but I try to persuade my students that it is actually easier to understand the methods rather than learn them by rote, since one small error results in a rote-learner failing the lab test and having to redo the whole test. The subtle differences between some methods, such as "`minVal`" and "`minPos`" or "`replaceAll`" and "`replaceOnce`" make rote learning difficult.
- Having students write their own code does not necessarily result in a student understanding their own code, especially when passing grades are given to students whose code did not even approximate the functionality described in the specification. Recall the student quoted earlier, who described how to achieve a pass by "making the answer look correct".

9.8 Reflections on Teaching

The above example of `countElement` illustrates how I teach all 13 methods that perform iterative processes on arrays. Each method is introduced by having students trace code. I then generalize from the (possibly multiple) traces the students have done, to build a stage 3 understanding (i.e., a concrete operational understanding) of how the code works for all possible inputs.

Piaget's crucial observation about children was that they do not simply know less than adults, but that children think differently from adults. To incorporate Piaget's crucial observation into neo-Piagetian theory and applying it to programming leads to the following: the novice programmer does not only know less than the accomplished programmer, but the novice programmer thinks differently from the experienced programmer. However, teachers and textbooks typically describe code in terms that are only understood by novices at the third stage, the post-tracing/concrete operational stage.

A second source of failure of traditional approaches is the lack of recognition of the importance of code tracing. Many students spend a long time in the pre-operational stage, and transition only slowly to the concrete operational stage.

Tracing is a tedious, error-prone process. It is therefore not surprising that many novice programmers prefer to not trace. If necessary, novices at the sensorimotor stage should be forced to trace; that is how the sensorimotor novice learns. However, novices who have reached the concrete operational stage can reason about code without needing to trace and may even express disdain for tracing [6]. Those students should not be forced to trace, at least not until their nascent concrete operational skills have failed them.

In some of my earlier papers, I have proposed that the aim of introductory programming courses was to get students to the concrete operational stage. Since then, I have revised my thinking. At university, the higher achieving students may be concrete operational at the end of the first semester of programing. However, I suspect most students who just manage to pass an introductory programming course are pre-operational.

If there is any validity to the widely held intuition that the distribution of student grades is bimodal, then perhaps it is an artifact of the grading – if an exam requires a great deal of concrete operational reasoning (and the students' answers are graded accordingly), then students will divide into those who can reason that way, and those who are yet to reach that stage.

There has been some interesting work by others on related, but different, teaching approaches [11, 28, 39]. As I have not yet given those papers the thought deserved, I will refrain from commenting here, but instead merely recommend those papers to the reader.

9.9 The Simple View of Programming?

The "Simple View of Reading" [40-43] is a theory of how children learn to read natural language, in which it is argued that learning to read requires mastery of two skills: phonetic decoding and the ability to understand a story when it is read to them. I see a similarity (or at least an analogy) between the Simple View of Reading and how the combination of tracing skill and explaining skill leads to skill in writing programs, as illustrated in Figure 7.

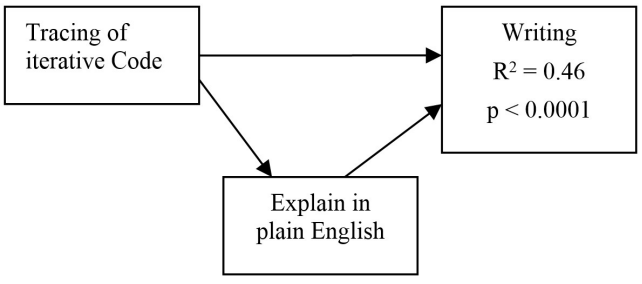

Figure 7: The upper portion of the model from Lopez et al. (2008).

The reader might object to the connection I am making between the Simple View of Reading and code writing – reading and writing are different activities. While I concede that point when it comes to the writing of complex code, I believe that there is a close connection between reading programs and writing programs for the novice programmer, at least in the first six months. Most novices at that early stage cannot write programs using the five-step problem solving process described by McCracken et al. Instead, writing code is an iterative two-step process, involving conjecture and justification. In the first step, the novice writes code that is a conjecture about what the code should be, while in the second step the novice uses tracing and explaining to establish whether or not the code actually does what the novice wants. Thus, for the early novice, reading code is an important component, perhaps the dominant component, of writing code. That is why I believe I am justified in making a connection with the Simple View of Reading. If, however, the reader is unconvinced by this paragraph's argument, they might at least consider the Simple View of Reading to be a useful analogy.

The connection with the Simple View of Reading suggests that predicting code writing ability via the linear combination of tracing and explaining is not strictly correct and is an artifact of the analysis method. Instead, the Simple View of Reading suggests that, for programming, the combination of tracing and explaining is a product, not a sum.

Unfortunately, making this connection to the Simple View of Reading leads to some pessimism. Among the people who study how children learn to read, there has for decades been a "reading war" (at least in the English-speaking world). On one side of the battle are the people who argue for the Simple View of Reading, or at least hold positions similar to it. On the other side are the people who argue for the "whole language" approach, which I will not describe here. If the people who study how children learn to read have been at war for decades, then there is little optimism that the people who teach computer programming will resolve their differences any time soon, given our community's penchant for arguing from intuition rather than research.

9.10 Are Tracing and Explaining All There Is?

The above speculation concerning a Simple View of Programming may give the false impression that I believe there is nothing more to competence in code writing than possessing the skills of tracing and explaining. As I enumerated earlier in this paper, McCracken et al. describe problem-solving as a five-step process: (1) Abstract the problem from its description, (2) Generate sub-problems, (3) Transform sub-problems into sub-solutions, (4) Re-compose, and (5) Evaluate and iterate. Clearly, those five steps involve skill in addition to tracing and explaining. I merely believe that students are not ready to begin learning those five-steps until at least their second semester of programming, after first acquiring the rudimentary coding skill that follows from learning tracing and explaining. In the literature on reading natural language, some authors advocate that children first need to pass through a phase of "learning to read" before they can proceed to "reading to learn". Similarly, I advocate that novice programmers need to pass through a phase of "learning to code" before they can proceed to "coding to learn" (i.e., learning problem-solving). I merely speculate that tracing and explaining are the most important skills, perhaps the only skills, required in that first "learning to code" phase.

The above speculation concerning a Simple View of Programming may give another false impression; that I believe the first "learning to code" phase should be a dry approach that focusses entirely upon code tracing and explaining and eschews code-writing. On the contrary, I believe the sentiment expressed in the first two pages of Lockhart's lament, about the traditional dry teaching of mathematics [44], applies equally to the teaching of programming – students should enjoy the first phase and they should write code. However, teachers should lower their expectations of how successful students will be at writing code in this first phase. Instead, teachers should use the student's clumsy attempts at code writing as an opportunity to motivate the teaching of tracing and explaining. Getting the balance right between code-writing and the other skills is part of the art of teaching. While it is true that students can be motivated by writing code, it is equally true that students can be de-motivated when they fail to succeed at code-writing. Explicitly teaching tracing and explaining will lead to a happier code-writing experience. Successfully teaching in the "learning to code" phase is not about teaching some skills to the exclusion of all others. Rather, it is about getting the balance right. Currently, there is an imbalance, with an over emphasis on code-writing.

10. CONCLUSION

I recently retired from my university appointment. What research I may do in the future remains to be seen, so I took the opportunity of this keynote to tell the story of my education research career. My apologies to the reader if telling my story seems self-indulgent, but humans learn best from stories. For tens of thousands of years, human have gathered around campfires and passed on knowledge through stories. For example, the aborigines of arid Australia tell creation myths that are entertaining and sometimes whimsical but embedded in the stories is knowledge for the children on where they can find water when, as adults, they eventually visit unfamiliar land. Through telling my own creation myth, I hope I have helped the reader to someday find water.

ACKNOWLEDGMENTS

This paper is an account of research carried out over twenty years. I thank all my education research collaborators over that time. I especially thank the people who changed the direction of my education research career: (0) Judy Kay, who had a very early effect on my teaching mindset, (1) my UTS colleague Jenny Edwards, who told me about Bootstrap (where would I be today had Jenny not knocked on my office door that day?), (2) my Bootstrap mentors Sally Fincher, Marian Petre and Josh Tenenberg, (3) my Leeds Working Group coauthors, especially Sue Fitzgerald, who among other things recruited half the group's members, (4) my BRACElet co-leaders Tony Clear and Jacqueline Whalley, especially Tony, who both instigated and sustained the project (5) Mike Lopez for his pivotal analysis of some BRACElet data, (6) my former Ph.D. student, "Doctor Donna" "Jean" Kingsbury (formerly Donna Teague), who accepted the risk of trusting my

early speculations on neo-Piagetian theory and who, through her think-aloud studies, developed those ideas substantially, and (6) Donna's think aloud subjects, most of all the initially baffling but eventually enlightening and inspiring "Donald", whoever he may be. To Beth Simon: working with you was fun and I wish we had done more work together, but we'll always have Pasadena. I also thank my partner, Ilona Box, for her support, her patience, her ideas, and her companionship over this long and sometimes bumpy journey. Finally, I thank Connie, Sam and Lulu.

REFERENCES

[1] D. P. Ausubel. 2000. *The Acquisition and Retention of Knowledge: A Cognitive View*. Kluwer Academic Publishers, ISBN 9780792365051.

[2] John B Biggs and Kevin F. Collis. 1982. *Evaluating the quality of learning: the SOLO taxonomy* (*structure of the observed learning outcome*). Educational psychology series. New York: Academic Press. ISBN 0120975505.

[3] Bootstrapping. 2002. Bootstrapping Research in Computer Science Education. 2002. http://depts.washington.edu/bootstrp/ (accessed 2020).

[4] R. Brooks .1983. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*. Volume 18, Issue 6, June 1983, 543-554.

[5] T. Clear, J. Whalley, P. Robbins, A. Philpott, A. Eckerdal, M. Laakso, & R. Lister. 2011. Report on the final BRACElet workshop. *Journal of Applied Computing and Information Technology*, 15(1). Retrieved September 28, 2020 from http://citrenz.ac.nz/jacit/JACIT1501/2011Clear_BRACElet.html

[6] Kathryn Cunningham, Rahul Agrawal Bejarano, Mark Guzdial, Barbara Ericson. 2020. "I'm not a computer": How identity informs value and expectancy during a programming activity. *International Conference of the Learning Sciences* (ICLS), Nashville, pp. 705-708.

[7] Hubert Dreyfus (1992) What Computers Still Can't Do: A Critique of Artificial Reason. MIT Press. ISBN 978-0262540674.

[8] Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, Marian Petre (2005) Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs. *Proceedings of the International Computing Education Research Workshop, Seattle, USA*. pp. 111-121. http://doi.acm.org/10.1145/1089786.1089797

[9] David Ginat. 2007. Hasty Design, Futile Patching and the Elaboration of Rigor. 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'07), Scotland. pp. 161-165. http://dx.doi.org/10.1145/1268784.1268832

[10] John Haugeland. 1985. *Artificial Intelligence: The Very Idea*. MIT Press. ISBN 978-0262580953.

[11] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. *In 2019 ITiCSE Working Group Reports* (*ITiCSE-WGR '19*), July 15–17, 2019, Aberdeen, Scotland UK. ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3344429.3372501

[12] F. S. Keller (1968) Goodbye teacher. *Journal of Applied Behavior Analysis*, 1 (Spring), pp. 79-89.

[13] Rudyard Kipling (1902) Just So Stories: The Original 1902 Edition with Illustrations by Rudyard Kipling. Suzeteo Enterprises (2019). ISBN 978-1645940166

[14] Raymond Lister, Elizabeth S. Adams, Sue C. Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate E Sanders, Otto Seppälä, Beth Simon, Lynda A Thomas (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGSCE Bulletin*, 36(4), 119-150. https://dl.acm.org/doi/10.1145/1041624.1041673

[15] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, Christine Prasad (2006) Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin* 38 (3), 118-122. https://dl.acm.org/doi/10.1145/1140124.1140157

[16] Raymond Lister, Colin Fidge, Donna Teague (2009) Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin* 41 (3), 161-165. https://dl.acm.org/doi/10.1145/1562877.1562930

[17] Raymond Lister, Tony Clear, Simon, Dennis J. Bouvier, Paul Carter, Anna Eckerdal, Jana Jackova, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, Errol Thompson (2009) Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bulletin*, Vol. 41, No. 4 (December), pp. 156-173. https://dl.acm.org/doi/10.1145/1709424.1709460

[18] Raymond Lister and Jenny Edwards (2010) *Teaching Novice Computer Programmers: bringing the scholarly approach to Australia*. Australian Learning and Teaching Council. 62 pages. ISBN: 1-921856-02-5 https://altf.org/wp-content/uploads/2016/08/Edwards_J-Lister_R_Associate-Fellowship_Final-Report-2010.pdf

[19] Raymond Lister (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Thirteenth Australasian Computer Education Conference, Perth*. https://dl.acm.org/doi/10.5555/2459936.2459938

[20] Raymond Lister (2016) Toward a Developmental Epistemology of Computer Programming. *Keynote paper/address at Workshop in Primary and Secondary Computing Education* (*WiPSCE*)*, Münster, Germany*, 13 - 15 Oct 2016. pp. 5-16. https://doi.org/10.1145/2978249.2978251

[21] Mike Lopez, Jacqueline L. Whalley, Phil Robbins, Raymond Lister (2008) Relationships between reading, tracing and writing skills in introductory programming. *Fourth International Workshop on Computing Education Research* (*Sydney, Australia, September 6 - 7*). ICER '08. ACM, New York, NY, 101-112. https://dl.acm.org/doi/10.1145/1404520.1404531

[22] Abraham H. Maslow (1966). *The Psychology of Science: A Reconnaissance*. Harper Collins. ISBN 978-0060341459.

[23] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda A Thomas, Ian Utting, Tadeusz Wilusz (2001) A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bulletin*, **33**(4):125-140. https://dl.acm.org/doi/10.1145/572133.572137

[24] Walter Milner (2008) A Loop is a Compression. *20th Annual Workshop of the Psychology of Programming Interest Group, Lancaster*. https://www.ppig.org/files/2008-PPIG-20th-milner.pdf

[25] S. Morra, C. Gobbo, Z. Marini, R. Sheese. (2008) *Cognitive development: neo-Piagetian perspectives*. New York: Lawrence Erlbaum Associates. ISBN 978-0805863505.

[26] Roy Pea. (1986) Language-Independent Conceptual "Bugs" in Novice Programming. *J. Educational Computing Research*. Vol. 2(1). pp. 25-36.

[27] M.A. Rubio (2020) Automatic Categorization of Introductory Programming Students. In: Martínez Álvarez F., Troncoso Lora A., Sáez Muñoz J., Quintián H., Corchado E. (eds) International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2019) and 10th International Conference on European Transnational Education (ICEUTE 2019). CISIS 2019, ICEUTE 2019. *Advances in Intelligent Systems and Computing, vol 951*. Springer, Cham. https://doi.org/10.1007/978-3-030-20005-3_31

[28] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teachers' Experiences of using PRIMM to Teach Programming in School. *In Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (*SIGCSE '19*), February 27-March 2, 2019, Minneapolis, MN, USA. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287477

[29] Siegler, R. S. (1996). *Emerging Minds*. Oxford, Oxford University Press. ISBN 978-0195126631.

[30] Elliot Soloway (1986) Learning to Program = Learning to Construct Mechanisms and Explanations. *Comm. ACM 29, 9* (*September 1986*), 850-858. https://doi.org/10.1145/6592.6594

[31] Juha Sorva. 2012. Visual program simulation in introductory programming education. (Doctoral dissertation). Aalto University, Espoo, Finland. ISBN (printed) 978-952-60-4625-9. https://aaltodoc.aalto.fi/handle/123456789/3534

[32] Donna Teague. 2015. *Neo-Piagetian Theory and the Novice Programmer*. Ph.D Thesis. Queensland University of Technology. http://eprints.qut.edu.au/86690/1/Donna_Teague_Thesis.pdf

[33] Lynda Thomas, Mark Ratcliffe, Benjy Thomasson. 2004 Scaffolding with object diagrams in first year programming classes: some unexpected results. *SIGCSE Bull*. 36, 1, pp 250-254. http://doi.acm.org/10.1145/1028174.971390

[34] Des Traynor, Susan Bergin, J. Paul Gibson. 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

[35] Anne Venables, Grace Tan, Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *International Computing Education Research Workshop (ICER), Berkeley, California, August 10-11*, 117-128. http://doi.acm.org/10.1145/1584322.1584336

[36] Vygotsky, Lev (1978). *Mind in Society*. London: Harvard University Press.

[37] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins Phil, P K Ajith Kumar, Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. *Proceedings of the 8th Australasian Conference on Computing Education*, 243–252. https://dl.acm.org/doi/pdf/10.5555/1151869.1151901

[38] Jacqueline L. Whalley, Tony Clear & Raymond Lister. 2007. The Many Ways of the BRACElet Project. Bulletin of Applied Computing and Information Technology 5, 1 (June). https://opus.lib.uts.edu.au/bitstream/10453/5747/1/2006009570.pdf

[39] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, Andrew J. Ko. 2019. A theory of instruction for introductory programming skills, Computer Science

Education, https://www.tandfonline.com/doi/full/10.1080/08993408.2019.1565235
[40] Philip Gough and William Tunmer (1986) Decoding, Reading, and Reading Disability Remedial and Special Education 7(1), 6-10.
[41] Wesley Hoover and Philip Gough (1990) The Simple View of Reading. Reading and Writing 2, 127-160. https://doi.org/10.1007/BF00401799.
[42] Baker, S.K., Fien, F., Nelson, N. J., Petscher, Y., Sayko, S., & Turtura, J. (2017). Learning to read: "The simple view of reading". Washington, DC: U.S. Department of Education, Office of Elementary and Secondary Education, Office of Special Education Programs, National Center on Improving Literacy. http://improvingliteracy.org (retrieved December 2020)
[43] Linda Farrell, Michael Hunter, Marcia Davidson, and Tina Osenga. The Simple View of Reading. https://www.readingrockets.org/article/simple-view-reading (retrieved December 2020)
[44] https://www.maa.org/external_archive/devlin/LockhartsLament.pdf (retrieved December 2020)