Toward a Developmental Epistemology of Computer Programming

Raymond Lister
University of Technology Sydney
Australia
raymond.lister@uts.edu.au

ABSTRACT

This paper was written as a companion to my keynote address at the 11th Workshop in Primary and Secondary Computing Education (WiPSCE 2016). The paper outlines my own research on how novices learn to program. Any reader whose interest has been piqued may pursue furher detail in the papers cited. I begin by explaining my philosophical position. In making that explanation, I do not claim that it is the only right position; on the contrary I allude to other philosophical positions that I regard as complimentary to my own. The academic warfare between these positions is pointless and counterproductive — all the established positions have something positive to offer. Having established my position, I then go on to argue that the work of Jean Piaget, and subsequent neo-Piagetians, offers useful insight into how children learn to program computers.

CCS Concepts

Social and professional topics → Computing education

Keywords

Neo-Piagetian theory, novice programmers.

1. INTRODUCTION

These are exciting times for computing education. Information technology is a ubiquitous part of life and a major driver of future economic growth. Accordingly, today's children need to be educated so that they can use computers in more sophisticated ways than their parents. Computer programming is seen as a central skill that distinguishes those who produce software from those who consume software. Consequently, the governments of many countries are developing computing curricula that will teach aspects of programming to all students, and will do so from the early years of schooling.

In drawing up these computing curricula, governments have relied upon computing academics as central participants. That academics are central to this curriculum design is at odds with the reality of computing education in universities—academics have not been tremendously successful in teaching computer programming to their own undergraduates. Why should there be any confidence that academics have something essential to contribute to the curricula for how children are taught computer programming?

One might characterize national efforts to develop computing curricula as collaboration between two groups, where each group understands half the problem. Academics understand computing and computer programming, but they do not understand children, teaching and learning. Teachers understand children, teaching and learning, but they do not appreciate the potential of computing and computer programming. In other words, academics understand the knowledge and capabilities students should have at the culmination of the educational experience, but not the path to that destination. Teachers have a better understanding of educational paths, but little understanding of the destination for computing. What is required is a way to bridge that gap between academics and teachers so they can work together productively.

This paper sketches my proposal for how the gap between academics and teachers can be bridged. Central to this paper is the concept of *developmental epistemology*, which is a concept from the work of Jean Piaget. Piaget preferred the term *genetic epistemology*, but the contemporary understanding of "genetics" often leads to confusion; perhaps the expression "*the genesis of knowledge*" better expresses to the contemporary reader what he intended. Developmental epistemology bridges the gap between developmental psychology and epistemology in general [27]. In developmental epistemology, the path for acquiring that knowledge is central. Developmental epistemology differs from developmental psychology (or child psychology) in that the knowledge of a specific domain is central. That is, the development of a child is not described in knowledge-independent terms, but instead is described in terms of how the child reasons within a specific knowledge domain.

Piaget's crucial observation was that children do not simply know less than adults, or that children believe things that are wrong. Instead, children think differently from adults. Adults, as part of acquiring their mature way of thinking, forget how they thought as children. Consequently, adults (including academics) inexperienced in teaching children communicate their knowledge to children in ways that children are not yet ready to understand. To children (and even undergraduates) computing academics speak a foreign and unknown language. Computing academics need to see themselves as working in partnership with those who can speak the student's language—teachers.

2. Positioning My Work

My research on novice programmers is often misunderstood. The single major criticism is that my work advocates a dull, dry and even non-constructivist pedagogy for programming. This companion paper to a keynote address is a unique opportunity to

WiPSCE '16, October 13-15, 2016, Münster, Germany
© 2016 ACM. ISBN 978-1-4503-4223-0/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2978249.2978251

record the thinking behind my work. In essence, I make no claim that my research implies a complete programming pedagogy. Nor should my research be criticized as if it is a complete pedagogy. To do so is to commit an error of reasoning known as a category error, in which the part is confused with the whole.

2.1 Phonics & Whole Language

Much of my own research work has been on how students read and understand programs, rather than how students write programs. Consequently, I have drawn inspiration from the extensive literature on how children learn to read their first natural language. In that literature, there has long been a debate on whether children should be taught to read via the phonics method or via the whole language method. Phonics is a bottom-up approach that emphasizes the relationship between written letters and spoken words. Whole language is a top-down approach, where children read texts early, for enjoyment, and are encouraged to make guesses about words they do not recognize, based on clues such as the context of the word in the text. Two New Zealand women of international influence are my icons for these two approaches—Marie Clay for the bottom-up phonics approach [5] and Sylvia Ashton-Warner for the top-down whole language approach [2]. Both of my icons write in a way that is accessible to a reader who is not a specialist in their field—I recommend them both.

I cannot claim to speak with authority about the teaching of reading, but I am not alone in believing that both the bottom-up and top-down approaches should be used to teach reading to every child. (I suspect if you watch a good teacher at work for sufficiently long enough, you will see the teacher blend the two methods, irrespective of which method is officially sanctioned at that school.) Tolstoy's novel "Anna Karenina" begins thus: *Happy families are all alike; every unhappy family is unhappy in its own way*. Likewise, both the bottom-up phonics approach and the top-down whole language approach can produce happy competent readers. However, when either method fails, they produce different types of unhappy incompetent readers. The combination of the two approaches is more likely to produce good readers.

The computing education community has an analogous debate. Some of us in this community are drawn to a top-down approach, and those people build wonderful programming environments, such as Scratch, that allow children to enjoy programming. I liken those people to the whole language advocates, and I therefore call them the "whole program" people. But there has also been a long bottom-up tradition in computing education of trying to structure the learning of programming in a way analogous to phonics. My chosen research area is the "phonics" of programming. (My dream is to be the Marie Clay of computer programming.) But here's the thing: we need both the top down approach and the bottom up approach. I like a comment in Juha Sorva's PhD thesis [28]. After surveying the literature on learning to program from varying warring academic camps, he ends with a reflective section entitled, "*Can We All Just Get Along*?"

I do have a problem with those in the "whole program" community who downplay the legitimacy of the "phonics of programming" approach. I believe they do so because they believe in extreme forms of social constructivism.

2.2 Social & Cognitive Constructivism

It is the year 2016, and we are all constructivists now. Sadly, however, there is another academic turf war to be had over exactly which form of constructivism is best – social constructivism versus Piaget-derived cognitive constructivism.

Social constructivists trace their intellectual roots to Vygotsky [40]. They believe that learning is socially situated and knowledge is constructed through interaction with others. Social constructivism does capture an important aspect of how people learn. Social constructivism has a large contribution to make to computing education.

But the danger with social constructivism (as with any ~ism) is when it is expressed in extreme forms that deny the legitimacy of any other perspective. In social constructivism's extreme form, terms such as "authentic assessment" are frequently used. (That's a clever rhetorical tactic; try arguing in favor of inauthentic assessment.) In extreme social constructivism, the quality of the learning environment is to be measured primarily by the degree to which the learning environment mimics the "real world", and the degree to which students show enthusiasm for what they are required to do. In this extreme form, social constructivism becomes a form of neo-behaviorism, where there exist only the actions and utterances exchanged between people; there is no notion of an individual's internal mental state. My objections would be misplaced—most schools of thought are prone to extremism—were it not unfortunately the case that extreme social constructivism is dominant; indeed rampant.

Extreme social constructivists claim that the debate is over, that social constructivism is the one and true constructivism. It is with reluctance that I must dispute that point. My reluctance is because I do not wish to be misconstrued as being opposed to social constructivism. I am merely against extreme social constructivism that admits no other perspectives on human learning. I won't attempt a rebuttal in this paper. Instead I will merely direct the reader to some of the dissenting literature. A notorious critique of extreme constructivism was authored by Kirschner, Sweller and Clark [15]. While that paper does make some very good points, it does tend to argue against extreme social constructivism from the opposite extreme, when neither extreme is right. A more balanced collection of essays was edited by Tobias and Duffy [39].

2.3 Three Streams of Educational Ideology

Kohlberg and Mayer [17] identified three "*Streams of Educational Ideology*". I believe that each of the three streams captures a worthwhile view on aspects of education, but none of the streams captures the entire picture. The streams are:

- **Romanticism:** According to this ideology, children should be given the freedom to learn through interacting with the world. This is the natural ideological home of the "whole of program" people and social constructivists. One of the original sources for this stream was Rousseau, in his treatise on education, entitled "Emile". The term "romanticism" is derived the philosophy promulgated by Rousseau — "romanticism" is not intended as a pejorative term.
- **Cultural Transmission:** In this ideology, the educator's job is the direct instruction of students in a body of knowledge. My own work is sometimes mistaken as belonging primarily to this category.
- **Progressivism:** This ideology is like romanticism in that it advocates that a child be given the freedom to learn through "natural interactions". Unlike romanticism, however, the development of the child is not the unfolding of an innate pattern, but instead the child is presented with a series of

learning environments that guide the child and reflect the child's progression through developmental stages. While Kohlberg and Mayer (being American) give much credit for the origins of this ideology to Dewey (hence "progressive"), it is an ideology that also has its origins with Piaget, and thus it is the natural home of cognitive constructivists, such as I.

As before, I believe all three ideologies have a contribution to make to the practice of education. It is merely that, as a researcher, I can only realistically work within one of those ideologies. My choice to do research in one of the three ideologies is not necessarily a rejection of the other two ideologies. As a researcher, I trust teachers to weave my ideology with the other ideologies into something that works in the classroom.

2.4 Education Research & Teaching Practice

I am fond of the following quote. While it is taken from a non-educational context, it captures for me the difference between the academic researcher and the practicing teacher:

An idea can tolerate a number of abstractions. Reality, on the other hand, must tolerate a number of contradictions.

— Kim Mahood [23]

I am an academic and a researcher. As part of my research I make abstractions, or simplifications, of the reality of teaching programming to children. It is the practicing teacher who has to take my abstractions, and also the abstractions of other researchers, which sometimes contradict mine, and weave it all together into something that works in the classroom. As I wrote earlier, I suspect if you watch a good teacher at work for long enough, you will see the teacher weave into a complete pedagogy the contradictory abstractions of different researchers.

3. THE STAGES OF PROGRAMMING

My phonic–cognitive constructivist–progressive oriented research on the teaching of programming has been heavily influenced by the work of Jean Piaget. What has emerged from my work, which was often done in collaboration with Donna Teague and others [1, 6, 7, 20, 29–37], is a four-stage model of learning to program. Those stages reflect Piaget's developmental stages. These four stages are (from least mature to most mature):

- **Sensorimotor:** The novice programmer has an incoherent understanding of program execution. When describing this stage, I sometimes refer to it as the **Pre-Tracing** stage.
- **Preoperational:** The novice can reliably manually execute ("trace") multiple lines of code. These novices often make inductive guesses about what a piece of codes does, by performing one or more traces, and examining the relationship between input and output. I sometimes refer to this as the **Tracing stage**.
- **Concrete operational:** The novice programmer reasons about code deductively, by reading the code itself, rather than using the preoperational inductive approach. This stage is the first stage where students show a purposeful approach to writing code. We sometimes refer to this as the **Post-Tracing stage**, and also the **Abstract Tracing stage** [36].
- **Formal operational:** This is the ultimate stage of Piagetian reasoning. It is the level at which the expert performs. A person thinking at the formal operational stage can reason logically, consistently and systematically. Formal operational reasoning also requires a reflective capacity — the ability to think about one's own thinking, including hypothetico-deductive reasoning, which is essential for debugging complex software. As this stage is expert thinking, and not to be expected in school children, it will not be discussed any further in this paper.

3.1 Piaget and the neo-Piagetians

Jean Piaget's work is not in fashion. There are many reasons for this, but the overall reason is that Jean Piaget worked in that early–to–mid 20th century intellectual period when structuralism was dominant, and so his direct legacy is not well received in this current period of post-structuralism. Another reason is that much of Piaget's writings about babies did not stand up to testing in recent experimental work with babies. That work on babies had the effect of decreasing the perceived credibility of Piaget's work with school-age children. That decrease is unfortunate, as Piaget's work with school-age children had a much better grounding in data than his work with babies.

Since Piaget's death, the "neo-Piagetians" have further developed Jean Piaget's work, improving the compatibility with both post-structuralism and with observational data [9, 10, 12, 22, 18, 26]. Both Jean Piaget's "classical" theory and neo-Piagetian theory describe cognitive development in terms of sequential, cumulative stages. However, neo-Piagetian theory differs in several ways, which are summarized very briefly in Table 1. For the purposes of this paper, it is necessary to further discuss two of those differences, which I do below.

3.1.1 Overlapping Waves

Like the neo-Piagetians, I do not classify novice programmers as being at a unique stage of development at any given moment. Nor do I believe that novice programmers make a sudden leap from one stage to the next. Rather than such a "staircase" model of development, I accept the neo-Piagetian "overlapping waves" model [3, 4, 26], as illustrated in Figure 1. In the overlapping waves model, a person exhibits a changing mix of the Piagetian stages. When commencing learning in a new knowledge domain, a person reasons predominantly at the sensorimotor stage, but as they learn they reason less at that stage and more at the preoperational stage, and so on to later stages. Thus multiple ways of reasoning coexist, but at a given moment a specific stage is often most obvious.

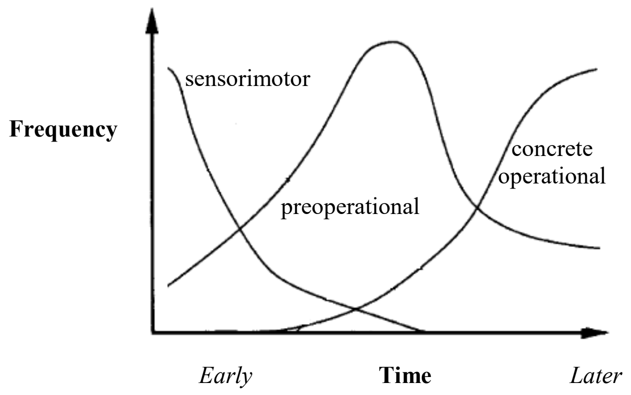


Figure 1: The Overlapping Waves Model

Later in this paper, I will use as shorthand terms such as "the sensorimotor programmer", "the preoperational programmer", and so on. When I do so, it should be understood that a stereotype is being invoked. Over a short period of time, a real student often exhibits a mix of the stages, and does not fit a stereotype.

Table 1: Classical versus Neo-Piagetian Theory

Classical Piagetian Theory by Jean Piaget	Vs.	Neo-Piagetian Theory
Is concerned with the general cognitive development of children.	vs.	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*.	vs.	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.
Suggests typical age ranges for each Piagetian stage, but empirical evidence shows great variation in age ranges.	vs.	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 "stair case" metaphor.	vs.	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. See Figure 1.

3.1.2 Accommodation and Assimilation

As the novice programmer learns, there are periods of time where the novice maintains their existing mix of Piagetian stages, even when the novice is taught something new. For example, having learnt about integer variables, a novice programmer may learn about floating point variables, without that affecting how they reason. 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., preoperational dominance) but that way of tracing fails when the student is introduced to loops. The novice must now change their way of reasoning about code. That change can lead to the novice going back to relying more heavily on earlier Piagetian stages. For example, the novice programmer must first understand how loops work (i.e., going back to sensorimotor dominance), then devise a new way of recording a trace on paper (i.e., back up to preoperational dominance). In Piagetian terms, this is known as *accommodation.* Figure 2 illustrates cycles of assimilation and accommodation over time.

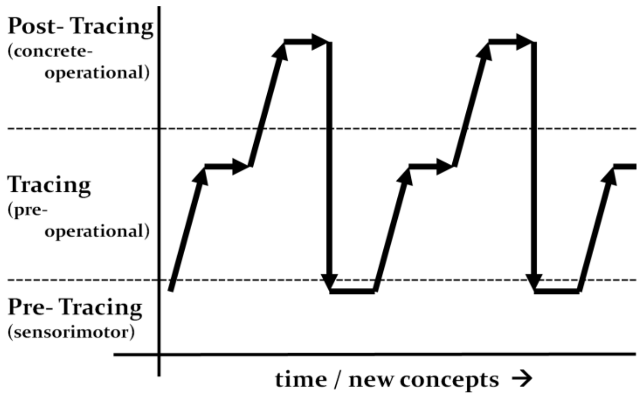


Figure 2: Assimilation and Accommodation

4. SENSORIMOTOR STAGE

At the least mature Piagetian stage, the sensorimotor stage, a novice programmer has difficulty tracing (i.e., manually executing) code. In the absence of a cognitive model of how a program works, sensorimotor programmers are (mis-)guided by perceptual superficialities that are ignored by better programmers. They tend to see static text, not a dynamic program.

There is an extensive non-Piagetian literature on the misconceptions of novice programmers [e.g. 11]. While that literature is essential reading for understanding the novice programmer, that literature under-represents some issues about novice programmers that only become clear 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 routinely swaps between conceptions, correct or otherwise, based on superficial aspects of the code that happen to be in view at the time. Consider the assignment statement. A sensorimotor programmer may change between assigning values left-to-right and right-to-left. Why should a sensorimotor programmer believe the '=' sign always means the same thing when some symbols in programming (e.g., '*') change meaning between contexts?

Given that the sensorimotor programmer has a very low commitment to conceptions of how programs work, they tend not to learn from hearing their teacher utter a general principle. That utterance is necessary, but it is not sufficient. Instead, the sensorimotor programmer learns a general principle by the embodied act of tracing code, and receiving feedback on their tracing, perhaps multiple times. The utterance of the general principle is most useful to the sensorimotor programmer when the teacher is correcting that student's answer.

The very idea of a machine is becoming an increasingly foreign concept for children in developed nations. The expression "*knowing what's under the hood*" is outdated—understanding an automobile is increasingly opaque to everyone except a specialist. The lack of intuition about the machine-like nature of computers is exacerbated by the forgiving and "intelligent" nature of user interfaces—word processing software corrects our spelling and search engines make suggestions about what search term we really meant to type. Sensorimotor programmers tend to think that an intelligent entity lurks inside their computer. Pea [25] referred to this as a type of "superbug", but I prefer to refer to it as the "*fallacy of the homunculus*". In one of my papers coauthored with Donna Teague [32] a student articulated the difficulty he experienced when he traced code:

> *... it takes me a very long time to remember how to think like a computer ... my mind wants to try and handle it a different way – but I'm like "No, a computer! You go line by line" ... but to me that's not the first way my mind wants to work ..."*

Figure 3 provides examples of questions on variables and assignment intended to move the sensorimotor programmer to the preoperational stage. (In this figure, and other figures like it, it should be understood that novice programmers may need many exercises like those shown in the figure. Most students do not progress from one Piagetian stage to the next by simply doing the small number of exercises shown in these figures.)

In Figure 3, questions 1(a) and 1(b) examine whether a student understands the basic semantics of assignment statements—that is, the value on the right of the assignment is copied to the left, overwriting the previous value. A well known programming misconception is caused by teachers likening a variable to a "box", which can lead to the misconception that a variable may hold more than one item; just as a box can [11].

In Figure 3, question 1(c) examines whether a student understands a sequence of assignment statements. The lines "`q = p`" and "`p = q`" may lead the novice susceptible to the homunculus fallacy to believe that this code swaps the values in "`p`" and "`q`".

In Figure 3, questions 1(d) and 1(e) test whether a student can track the changing values in variables. Even when students understand assignment statements and also the effect of a sequence of statements, they often make errors because they cannot reliably track the changing values in variables through a series of assignment statements. In many cases this is because students try to retain the variable values in their heads (i.e., working memory), rather than writing down those values [19]. Even when they do use pen and paper, sensorimotor programmers use *ad hoc* error prone ways of recording variable values. For the sensorimotor programmer, every piece of code is an opportunity to devise a new way of recording variable values on paper.

Figure 4 illustrates broadly the same type of questions as Figure 3, but for arrays. As discussed earlier, sometimes the neo-Piagetian stage of a novice programmer will drop when the novice encounters a new programming concept. Arrays (and loops) are an obvious case where this happens. When it happens, basic ideas have to be revisited. It may seem initially odd to an experienced programmer, but why should a novice believe that principles that apply to scalar variables will also apply to array variables?

4.1.1 Arrays with Constant Subscripts

In Figure 4, note that the array subscripts are all constants. I have found that introducing arrays with constant subscripts familiarizes students with basic array concepts before I then introduce variables as subscripts. I have found that students (albeit university undergraduates) have almost no more difficulty on arrays with constant subscripts than they do with scalar variables. (With university undergraduates I have sometimes introduced arrays with constant subscripts in the same lecture where I introduce scalar variables.) The problems students have with arrays begin when I introduce variables as subscripts.

5. PREOPERATIONAL STAGE

The preoperational novice can trace code with some accuracy. However, a preoperational programmer tends not to abstract from the code itself—there is nothing but the code. The only way that a preoperational programmer can reason about a piece of code is by tracing that code.

Because all they can do is trace code, preoperational programmers reason by induction. That is, the preoperational 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 5, the variables names "`y1`", "`y2`" and "`y3`" invite the preoperational 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`. Given that single input/output pair, some preoperational programmers answer incorrectly that the purpose of the code is to reverse the order of the values in the three variables [29, 31, 32].

A novice programmer who traces the code in Figure 5 only once and arrives at the wrong answer is an early preoperational programmer. In that early phase, the effort of tracing code is great and so the novice tends to only perform a single trace. As dexterity at tracing improves, the preoperational novice becomes more willing to perform more than a single trace. In one of Donna Teague's case studies [32], in which two students "Lucas" and "Sierra" worked together on the problem in Figure 5, Sierra jumped to the wrong conclusion after a single trace (i.e. Sierra was early preoperational), but Lucas performed a second trace with different initial values, and arrived at the correct answer. (While Sierra arrived at the correct answer, he did so by tracing, and is therefore preoperational; just not early preoperational.)

The preoperational novice uses that same inductive approach when attempting to debug their own code. That is, the preoperational novice traces their buggy code with specific values, and then makes 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 [14]. Using this strategy of "repeat−trace−patch−until−success", preoperational students may eventually succeed in producing correct solutions to small programming problems, but they will only do so after considerable time. It is a demoralizing way to write code. For this reason, preoperational programmers should only write code when closely supervised. Universities have traditionally taught programming by having students write a great deal of code. A challenge for teaching programming in schools will be to find a different way to teach preoperational programmers; a way that does not emphasize code writing. If school children who are preoperational programmers are forced to write a lot of code, eventually teachers will lead a push to remove programming from the school curriculum.

Since a preoperational programmer can only reason about code by tracing, they are not able to use scaffolding intended to help them with writing code. For example, Thomas, Ratcliffe, and Thomasson [38] 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'...*

The most important insight stemming from the application of Piagetian ideas to programming is the identification of the preoperational programmer, and that the preoperational stage is a natural stage of progression for a novice programmer—it is not an aberrant condition. Furthermore, students can spend a long time in the preoperational stage. Many students may only progress beyond the preoperational stage after being shown and led through many examples of code. I suspect that most children will still be working at the preoperational stage of programming when they commence high school, even if they have had a great deal of exposure to code before commencing high school.

When answering these questions, you may write your working out anywhere on the page outside the answer boxes.

Q1 (a) In the boxes, write the values in the variables after the following code has been executed:

```
int a = 1;
int b = 2;
a = 3;
```

The value in `a` is $\boxed{}$ and the value in `b` is $\boxed{}$

Q1 (b) In the boxes, write the values in the variables after the following code has been executed:

```
int r = 2;
int s = 4;
r = s;
```

The value in `r` is $\boxed{}$ and the value in `s` is $\boxed{}$

Q1 (c) In the boxes, write the values in the variables after the following code has been executed:

```
int p = 1;
int q = 8;

q = p;
p = q;
```

Beware the fallacy of the homunculus!

The value in `p` is $\boxed{}$ and the value in `q` is $\boxed{}$

Q1 (d) In the boxes, write the values in the variables after the following code has been executed:

```
int x = 7;
int y = 5;
int z = 3;

x = y;
z = x;
y = z;
```

The value in `x` is $\boxed{}$ the value in `y` is $\boxed{}$ and the value in `z` is $\boxed{}$

Q1 (e) In the boxes, write the values in the variables after the following code has been executed:

```
int a = 7;
int b = 3;
int c = 2;
int d = 4;

int e = a;

a = b;
b = e;
e = c;
c = d;
d = e;
```

The value in `a` is $\boxed{}$ the value in `b` is $\boxed{}$

The value in `c` is $\boxed{}$ the value in `d` is $\boxed{}$ and the value in `e` is $\boxed{}$

Figure 3: Sample questions on variables and assignment for sensorimotor to preoperational students.

When answering these questions, you may write your working out anywhere on the page outside the answer boxes.

Q2 (a) In the boxes, write the values in the array after the following code has been executed:

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

The value in `a[0]` is ☐ and the value in `a[1]` is ☐

… *Questions* Q2 (b) *and* (c) *omitted because of the paper page limit* …

Q2 (d) In the boxes, write the values in the array after the following code has been executed:

```
int [] x = {7, 5, 1};

x[0] = x[1];
x[2] = x[0];
x[1] = x[2];
```

The value in `x[0]` is ☐ the value in `x[1]` is ☐ nd the value in `x[2]` is ☐

Figure 4: Sample questions on arrays and assignment for sensorimotor to preoperational students.

If you were asked to describe the purpose of the code below, a good answer would be "*It prints the smaller of the two values stored in the variables* `a` *and* `b`".

```
if (a < b):
    print a
else:
    print b
```

In one sentence that you should write in the empty box below, describe the purpose of the following code.

Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code, like the purpose given for the code in the above example (i.e. "*It prints the smaller of the two values stored in the variables* `a` *and* `b`").

Assume that the variables y1, y2, and y3 are all variables with integer values.

In each of the three boxes that contain sentences beginning with "`Code to swap the values` …" assume that appropriate code is provided instead of the box – do **NOT** write that code.

```
if (y1 < y2):  Code to swap the values in
               y1 and y2 goes here.

if (y2 < y3):  Code to swap the values in
               y2 and y3 goes here.

if (y1 < y2):  Code to swap the values in
               y1 and y2 goes here.
```

Sample answer for WIPSCE paper:

It sorts the values into descending order

Figure 5: A question that tends to distinguish between preoperational and concrete operational programmers.

When answering these questions, you may write your working out anywhere on the page outside the answer boxes.

Q3 (a) In the boxes, write the values in the variables after the following code has been executed:

```
int x = 7;
int y = 5;
int z = 0;

z = x;
x = y;
 y = z;
```

The value in `x` is $\boxed{}$ the value in `y` is $\boxed{}$ and the value in `z` is $\boxed{}$

Q3 (b) In part (a) above, what do you observe about the final values in `x` and `y`? Write your observation (in one sentence) in the box below.

Sample answer for WIPSCE paper: *The values have swapped.*
(Any answer that conveyed that meaning would be satisfactory.)

The purpose of the following three lines of code is to swap the values in variables `a` and `b`, for any set of possible values stored in those variables. Assume that variables `a`, `b` and `c` have been declared and initialized.

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

Q3 (c) In one sentence that you should write in the box below, describe the purpose of the variable "c" in the above code.

Sample answer for WIPSCE paper: *Used to hold one of the other values temporarily*
(Any answer that conveyed that meaning would be satisfactory.)

Q3 (d) In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables. Assume that variables i, j and k have been declared and initialized.

```
j = i;
i = k;
k = j;
```

Sample answer for WIPSCE paper: *Swaps the values in i and k.* . (Any answer that conveys that meaning is fine, but the answer must indicate that it is variables i and k that are swapped.)

Q3 (e) Assume two integer variables "`first`" and "`second`" have been initialized.
Write code to swap the values stored in those two variables.

Sample answer for WIPSCE paper:

```
int temp   = first;                  int temp   = second;
    first  = second;    -- or --         second = first;
    second = temp;                       first  = temp;
```

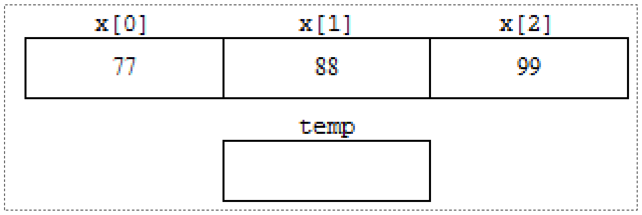
Figure 6: Sample questions on variables and assignment for preoperational to concrete operational students.

When answering these questions, you may write your working out anywhere on the page outside the answer boxes.

Q4 (a) The first line of the code below declares an array of integers "x" and initializes it to the three values 77, 88 and 99. The diagram in the dashed box represents that array "x" and its initial values, and also the variable "`temp`" declared in the second line of code. In the boxes below the code, write the values in the array "x" after all the code has been executed.

```
int [] x = {77, 88, 99};


int temp = x[0];
    x[0] = x[1];
    x[1] = x[2];
    x[2] = temp;
```



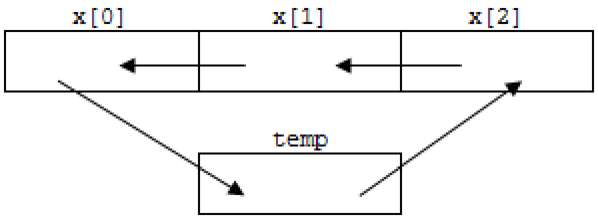
The value in `x[0]` is ☐ e value in `x[1]` is ☐ l the value in `x[2]` is ☐

Q4 (b) In part (a) above, what do you observe about the final values in the array "x" compared to their initial values? Write your observation (in one sentence) in the box below.

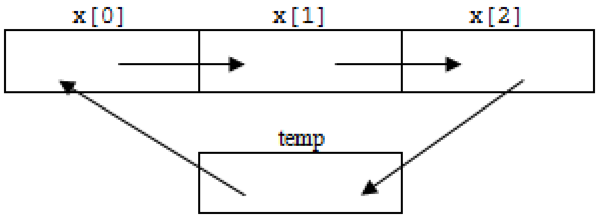
Sample answer for WIPSCE paper: *The values in the array "x" have moved one place to the* *<u>left</u>*, *with the leftmost element of "x" moving around to the rightmost position.* (Any answer conveying that meaning would be satisfactory.)

(**Note:** *The remaining questions would be withheld from students until parts (a) and (b) had been answered.*)

Q4 (c) The following picture shows the array "x" and the variable "temp" from questions (a) and (b). The arrows show how the data in "x" moved as each line of code was executed. Near each of those arrows, write the line of code from (a) represented by that arrow.



Q4 (d) The following picture represents the array "x" and the variable "temp", but this time the arrows show the data moving in the other direction, to the right, with the rightmost element of "x" moving around to the leftmost position. Near each of those arrows, write the line of code represented by that arrow. The lines of code needed are similar to but NOT the lines from part (a).



Q4 (e) In the box below, rewrite your code from part (d) in the correct order, one line of code under another like in part (a), so that when those lines are executed one after the other the values in the array are moved to the right, with the rightmost element of "x" moving around to the leftmost position. Do not bother rewriting the first line from part (a), "`int [] x = {77, 88, 99};`"

Figure 7: Sample questions on arrays and assignment for preperational to concrete operational students.

6. CONCRETE OPERATIONAL STAGE

It is only at the third Piagetian stage, the concrete operational stage, that the novice programmer begins to reason about abstractions of code. For example, it is only at this third stage that the novice programmer can use diagrams to productively reason about code.

In this paper, little else needs to be said about the concrete operational programmer. (Readers interested in learning more about the concrete operational programmer are referred elsewhere [1, 7, 20, 29, 31-32, 34-37].) The concrete operational programmer thinks the way we previously believed all novice programmers thought, prior to the neo-Piagetian perspective on programming. Therefore, the concrete operational programmer may further develop their programming skills using the approach used for decades — by having them write lots of code. The message all school teachers must internalize is that not all their students will transition quickly and easily to the concrete operational stage. The role of the teacher is to help students make the transitions between the Piagetian stages. The remainder of this section considers the transition between the preoperational and concrete operational stages, and how a teacher can facilitate it.

6.1 Assimilation and Accommodation

Figure 2 illustrates that progress through the Piagetian stages is not usually monotonic. Sometimes, when a new programming construct is introduced (e.g. loops) students can regress temporarily to lower Piagetian stage of reasoning. I do not believe that such a regression is necessarily bad. In fact, I believe teachers should aim to encourage it. To achieve that aim, teachers should seek to develop the higher Piagetian stages of reasoning with as few program construct as possible, and only then introduce the next programming construct. That aim is illustrated in Figure 6. Using only the programming constructs of variables and assignment, the questions in Figure 6 require the novice to reach the concrete operational stage. In Figure 6, questions 3(a) and 3(b) are questions answerable by a preoperational programmer. Question 3(a) requires the novice to trace code and 3(b) requires to the novice to make an inductive observation about that code. Questions 3(c) to 3(e) require concrete operational reasoning. Questions 3(c) and (d) examine whether a student understands some code, while question 3(e) requires the student to write similar code. Swapping the values in two variables like this is the "Hello World" of concrete operational reasoning [6].

6.2 Reversibility

One of the defining qualities of a concrete operational programmer is the ability to write code that reverses the effect of a given piece of code (i.e., when reversal is possible). Figure 7 provides an example, where the student is given code to move the elements of an array one place to the left, with the leftmost element moving to the rightmost position. In Figure 7, questions 4(a) and 4(b) are answerable by a preoperational programmer. Question 4(c) develops the student's understanding of a diagrammatic representation of the code, while questions 4(d) and 4(e) lead the student through writing code that reverses the effect of the given code.

(By the way, I have noticed that asking the students to copy array values from left to right can cause a regression to the sensorimotor stage, as some students are confused by the values in the diagram moving one way (i.e., left to right) while the assignment statements in the code move values the other way (i.e., from right to left).

The principles of accommodation and assimilation can be applied to this reversibility problem. When the student has later been taught variables as array subscripts and has also been taught loops, this problem can be revisited, for arrays of arbitrary size. (Interested readers are referred elsewhere [1, 7, 20, 29, 35, 37].)

6.3 An Aside on the Language Wars

For decades, computing academics have debated which language is best for teaching programming. The code shown Figures 6 and 7 could be from many languages. If the semicolons were deleted, that code could be from many other languages. I have found that many students in my programming classes take some time to come to grips with the material in Figures 6 and 7 — selecting the "best" programming language for teaching programming is an issue, but it is secondary to the issues discussed in this paper.

6.4 Getting to Abstract Tracing

In the section on the preoperational programmer, I described the early preoperational programmer as someone who is reluctant to perform more than one trace. Also, while it was not stated in that section, the early preoperational programmer tends to make *ad hoc* choices for the initial values of the trace. Later, as the preoperational programmer moves from the early phase to the middle-to-late phase, that student becomes willing to perform more than one trace, with different initial values intended to exercise different pathways through the code. Accordingly, a useful task a teacher may set is to have their students devise initial values that will exercise different pathways through a given piece of code.

The transition from preoperational to concrete operational begins when the novice begins to think of a single set of initial values as being representative of a class of possible initial values. For example, in some hypothetical piece of code, two variables "`a`" and "`b`" may be initialized to `a=1` and `b=2` to represent all possible initial values where `a < b`. To develop this type of thinking, one approach a teacher might use is to have several students generate specific initial values to exercise a prescribed path through a given piece of code and then lead a class discussion on the properties common to the answers provided by the students.

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 algebraic-like constraints on the possible values in each variable. For example, consider a student performing an abstract trace on the three "if" statements at the bottom of Figure 5. After the first of those `if` statements, the student would think of `y2` as holding any possible value that satisfies the condition that it is less than the value in `y1`. After the second `if`, the student thinks of `y3` as holding any possible value that satisfies the condition that it is less than the values in both `y1` and `y2`. Donna Teague and I have referred to this as "*abstract tracing*" [36]. A teacher might, after going through the steps I suggested in the previous paragraphs, illustrate an abstract trace to the students.

7. CONCLUSION

While these are exciting times in computing education, we are at the peak of inflated expectations in the hype cycle, and we are perhaps heading for a trough of disillusionment [13]. If the gap between academics and teachers is not bridged, then developmentally inappropriate computing curricula will be implemented. If the traditional academic view of computing is overly influential, then the current "experiment" in computing education may eventually be judged a failure, just as the "new math" of the 1960s was judged a failure [16]. If that happens, then computer programming in schools will eventually be "dumbed down" to something that teaches little of lasting value to children.

In the current excitement, it is often overlooked that ours is not the first attempt to introduce computer programming into schools. The 1980s was an equally exciting period, but it ended in failure. Of course, one important difference today is the ubiquity of computers, but I do not believe that ubiquity alone will lead to success this time. The lessons from the 1980s have not been learnt and we are therefore in danger of repeating that failure. What they did not have in the 1980s, and what we still do not have now, is an intimate understanding of the mental processes of the novice programmer.

Larry Cuban is an American education academic who over the decades has written several influential books about computers in schools. In his most recent book [8], Cuban used a metaphor to describe the disconnection between high level attempts to implement education change and the day-to-day reality in the classroom:

> "*… I wrote about the hurricane of school reform. … a hurricane whips up twenty-foot high waves, agitating the surface of the ocean … Yet deep down on the ocean floor, life goes on, undisturbed by the roiling waters and huge waves on the surface. I compared that ocean floor to the nation's classrooms, where both change and continuity unfold in regular, undisturbed patterns*." [8, pp 15-16]

The national curricula for computing in schools are our hurricane. Such high level design is essential. I do not seek to diminish its importance. However, if we do not also study intimately the wonderful and delicate creatures of the ocean floor, and design day-to-day teaching and learning activities that are sensitive to the growth of those creatures, at each and every stage of their growth, then the current wave of computing educational reform will dissipate, like the waves after a hurricane, leaving some future generation of computing educators to salvage the detritus.

ACKNOWLEDGMENTS

I thank my former Ph.D. student, Dr. Donna Teague, who accepted the risk of trusting my early ideas on neo-Piagetian theory and who developed those ideas substantially through her think-aloud studies of novice programmers. Donna and I are both grateful for the contributions made by Alireza Ahadi and Malcolm Corney. I thank my partner, Dr. Ilona Box, for her long term support through many difficult and frustrating times, as neither computing education research nor developmental epistemology are widely respected in academia; misunderstood and considered suspect by computing academics, by teaching & learning acolytes, and by university management alike. I thank the workshop organizers for providing me with a keynote platform on which to express my theories, my fears and my hopes.

REFERENCES

[1] Ahadi, A., Teague, D. and Lister, R. (2014). *Falling Behind Early and Staying Behind When Learning to Program*. Proceedings of the Psychology of Programming Interest Group Annual Conference 2014. Brighton, UK. 25-27th June. du Boulay, B. and Judith Good, J. (Eds) pp. 77-88. http://www.sussex.ac.uk/Users/bend/ppig2014/8ppig2014_submission_15.pdf

[2] Ashton-Warner, S. (1963) *Teacher*. Simon and Schuster.

[3] Boom, J (2015) *A new visualization and conceptualization of categorical longitudinal development: measurement invariance and change*. Frontiers in Psychology, Vol. 6. http://journal.frontiersin.org/article/10.3389/fpsyg.2015.00289/full

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

[5] Clay, M. (2005) *An observation survey of early literacy achievement*. Auckland, N.Z.; Portsmouth, NH: Heinemann.

[6] Corney, M., Lister, R., and Teague, D. (2011) *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, January. CRPIT, 114. pp. 95-104. http://crpit.com/confpapers/CRPITV114Corney.pdf

[7] Corney, M., Teague, D., Ahadi, A. and Lister, R. (2012). *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia. CRPIT, 123. pp. 77-86. http://www.crpit.com/confpapers/CRPITV123Corney.pdf

[8] Cuban, L. (2013) *Inside the Black Box of Classroom Practice*. Cambridge, Mass. : Harvard University Press.

[9] Demetriou, A. (Ed.) (1987) (A special double issue neo-Piagetian theory.) International Journal of Psychology, Vol. 22, Issues 5 & 6. ISSN: 0020-7594.

[10] Demetriou, A., Shayer, M. and Efklides, A. (Eds). (1992) *Neo-Piagetian Theories of Cognitive Development: Implications and Applications for Education*. London : New York: Routledge. ISBN 0415117496.

[11] Du Boulay, B. (1989). *Some difficulties of learning to program*. In Soloway, E. & Sphorer, J. (Eds), Studying the novice programmer. Lawrence Erlbaum. pp. 283-300.

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

[13] Gartner (2016) Gartner Hype Cycle. http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp Accessed August 2016.

[14] Ginat, D. (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

[15] Kirschner, P., Sweller, J. and Clark, R. (2006) *Why Minimal Guidance during Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*. Educational Psychologist, vol. 41, no. 2, pp. 75-86.

[16] Kline, M. (1973). *Why Johnny Can't Add: The Failure of the New Math*. New York: St. Martin's Press.

[17] Kohlberg, L. and Mayer, R. (1972) *Development as the Aim of Education*. Harvard Educational Review, Vol. 42, No. 4, pp. 449-96. Also in Smith, L. (Ed.) (1992) Jean Piaget: Critical Assessments, Volume 3, Chapter 59, pp. 277- 321.

[18] Levin, I. (Ed.) (1986): *Stage and Structure: Reopening the Debate*. Norwood, New Jersey: Ablex.

[19] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al (2004). *A Multi-National Study of Reading and Tracing Skills in Novice Programmers*. SIGSCE Bulletin, 36(4), 119-150.

[20] Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth

Australasian Computer Education Conference, Perth. http://crpit.com/confpapers/CRPITV114Lister.pdf

[21] Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). *Relationships between Reading, Tracing and Writing Skills in Introductory Programming*. Paper presented at the 4th Annual International Conference on International Computing Education Research (ICER 2008), Sydney, Australia. http://dx.doi.org/10.1145/1404520.1404531

[22] Lourenco, O. and Machado, A. (1996): In defense of Piaget's theory: a reply to 10 common criticisms. *Psychological Review* 103, **1**:143-164.

[23] Mahood, K. (2007) *Blow-ins on the cold desert wind*. Griffith REVIEW 15: Divided Nation. https://griffithreview.com/articles/blow-ins-on-the-cold-desert-wind/

[24] Morra, S., Gobbo, C., Marini, Z., and Sheese, R. (2008) Cognitive development: neo-Piagetian perspectives. New York: Lawrence Erlbaum Associates.

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

[26] Siegler, R. S. (1996). *Emerging Minds*. Oxford U. Press.

[27] Smith, L. (2009) *Piaget's Developmental Epistemology*, In Müller, U., Carpendale, J. and Smith, L. (Eds) The Cambridge Companion to Piaget. pp. 64-93.

[28] Sorva, J. (2012) Visual Program Simulation in Introductory Programming Education. Ph.D Thesis. Aalto University. http://lib.tkk.fi/Diss/2012/isbn9789526046266/isbn9789526046266.pdf

[29] Teague, D., Corney, M, Ahadi, A. and Lister, R. (2012). *Swapping as the "Hello World" of Relational Reasoning: Replications, Reflections and Extensions*. Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia. CRPIT, 123. pp. http://www.crpit.com/confpapers/CRPITV123Teague.pdf

[30] Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A. and 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*. Australasian Association for Engineering Education Conference (AAEE2012). http://www.aaee.com.au/conferences/2012/documents/abstracts/aaee2012-submission-22.pdf

[31] Teague, D., Corney, M., Ahadi, A. and Lister, R. (2013) *A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers*. Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia. CRPIT Vol. 136. pp. 87-96. http://crpit.com/confpapers/CRPITV136Teague.pdf

[32] Teague, D. and Lister, R. (2014) *Longitudinal Think Aloud Study of a Novice Programmer*. Sixteenth Australasian Computing Education Conference (ACE2014), Auckland, New Zealand. CRPIT Vol. 148. pp. 41-50. http://crpit.com/confpapers/CRPITV148Teague.pdf

[33] Teague, D. and Lister, R. (2014) *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Sixteenth Australasian Computing Education Conference (ACE2014), New Zealand. CRPIT Vol. 148. pp. 65-74. http://crpit.com/confpapers/CRPITV148Teague%20.pdf

[34] Teague, D., Ahadi, A. and Lister, R. (2014). *Programming: Reading, Writing and Reversing*. 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'14), Sweden. pp. 285-290. http://doi.acm.org/10.1145/2591708.2591712

[35] Teague, D., and Lister, R. (2014). *Blinded by their Plight: Tracing and the Preoperational Programmer.* Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG). Brighton, UK. pp. 53-64. http://www.sussex.ac.uk/Users/bend/ppig2014/6ppig2014_submission_8.pdf

[36] Teague, D. and Lister, R. (2015) *Mired in the Web: Vignettes from Charlotte and Other Novice Programmers.* Seventeenth Australasian Computing Education Conference (ACE2015), Sydney, Australia. CRPIT Vol. 160. pp. 165 – 174. http://crpit.com/abstracts/CRPITV160Teague.html

[37] Teague, D. (2015) *Neo-Piagetian Theory and the Novice Programmer*. Ph.D Thesis. Queensland U. of Technology http://eprints.qut.edu.au/86690/1/Donna_Teague_Thesis.pdf

[38] Thomas, L., Ratcliffe, M., and Thomasson, B. (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

[39] Tobias, S., and Duffy, T. (Eds.) (2009) *Constructivist instruction : success or failure*? New York : Routledge.

[40] Vygotsky, L. S. (1978). *Mind in society: The development of higher pyshcological processes*. Cambridge, MA: Harvard University Press.