Chapter 3

Microworlds, Objects First, Computational Thinking and Programming

Greg Michaelson
School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, Scotland

1. Overview

Teaching of programming has long been dominated by language oriented approaches, complemented by industrial design techniques, with little attendant pedagogy. An influential alternative has been Papert's constructivism, through playful exploration of constrained microworlds. The archetypal microworld is based on turtle graphics, as exemplified in Papert's Logo language. Here, students compose and repeat sequences of operations to steer and move a turtle that leaves a trail behind it. Contemporary graphical environments, like Alice and Scratch, augment the turtle world with colourful interacting animated avatars.

However, the microworld approach scales poorly to systematic programming driven by problem solving. Many students find the transition from novice coding to problem solving oriented programming problematic (Moors and Sheenan, 2017). Furthermore, microworld languages tend to be relatively impoverished, lacking types and data structures.

Objects First is a contemporary approach to teaching programming through object orientation, which seeks to bridge microworlds and systematic programming. Here, students explore, modify and extend pre-formed objects analogous to microworlds, in constrained subsets of full strength languages, typically Java. However, there is growing evidence that, as with the original microworlds, some students find the transition to problem solving based programming difficult.

Computational thinking (CT), as popularised by Wing, offers an approach to problem solving in which programming is the final stage. CT has been widely heralded as a new pedagogy of programming. However, interpretations of CT vary widely from a loose assemblage of techniques to a systematic discipline.

In this chapter, I will argue that both microworlds and Objects First build superficial programming skills at the expense of deeper competences in problem solving. I will further argue that systematic CT, driven by seeking patterns in concrete instances, offers a way to refocus on problem solving for programming.

What follows may seem a bit disjointed but it gets there in the end. My approach is a mix of pedagogy, history and opinion: I trust it’s clear which is which.

2. Microworlds and Logo

Since Wing’s highly influential intervention (2006) there has been worldwide interest in Computational Thinking (CT) as a pedagogy of problem solving. Tedre and

Denning (2016) offer a succinct account of CT before and after Wing. In particular, they draw attention to the key role of Papert in envisioning CT.

Papert (1993) was a proponent of Piaget’s constructivist model of cognitive development. Here, a child’s transition, from the pre-adolescent concrete to the adult abstract operational stages of though, is informed by learning by discovery, termed *bricolage,* that is the exploration of different assemblies of available skills. Thus, Papert expounded a notion of problem solving based on *combinatorial thinking*, i.e. systematic exploration, in some *microworld* i.e. a constrained domain, *through thinking about thinking* i.e. debugging an incorrect solution to find a better one.

This approach derives from early Artificial Intelligence research. In their report on activities in the MIT Ai laboratory, Minsky and Papert (1971) say that:

> “…we see solving a problem often as getting to know one's way around a "micro-world" in which the problem exists.” (Minsky and Papert ,1971) (cited in (Weir, 1987, p.12)).

and that:

> “*We think that learning to learn is very much like debugging complex computer programs.* To be good at it requires one to know a lot about describing processes and manipulating such descriptions.” (italics in original) (Minsky and Papert ,1971)

Hence, a notation for describing microworld processes, and an environment for manipulating process descriptions, could facilitate both combinatorial thinking and thinking about thinking.

Papert was highly enthused by the educational possibilities offered by mass access to computers with graphical capabilities. He envisaged how what we now call personal computing facilities might be used to embody and animate microworlds for teaching children.

Primarily concerned with mathematics education, Papert’s Logo programming language was intended for manipulating a geometric microworld of turtle graphics. His idea was that, by acting out the behaviours they wished the turtle to perform, a neophyte could learn how to assemble rules characterising those behaviours in the microworld on the computer. Rule assemblies could then be falsified by comparing the behaviour of the on-screen turtle with the intended behaviour, for reformulation.

Papert promoted the benefits of:

> “…*deliberately* thinking like a computer, according, for example, to the stereotype of a computer program that proceeds in a step-by-step, literal, mechanical fashion. There are situations where this style of thinking is appropriate and useful. Some children’s difficulties in learning formal subjects such as grammar or mathematics derive from their inability to see the point of such a style.”(italics in original) (Papert,1993, p.27).

While Papert is somewhat vague about his pedagogy, his collaborator Weir (1987) thoroughly explored how best to use computers to support "a kind of messing about". For Weir, a microworld is a:

> "*small, coherent domain of objects and activities implemented as a computer program* and corresponding to an interesting part of the real world" (italics in original)(Weir, 1987, p.12)

She went on to elaborate the key features of computer based environments for exploring a microworld, of which I think the following are the most pertinent:

> "1. …as the learner interacts with environment (sic), she "manipulates" concrete embodiments of the concepts to be learned; she "experiences" the concepts directly, not through language about them.
>
> 2. … The idea is to juxtapose experience in the real world with experience in the computational world, so that each complements the other.
>
> 3. A cluster of activities is made available so that any one concept is met in several different contexts and in different combinations.
> …
> 5. It is useful to choose aspects of the environment about which the learner is likely to have intuitions, to have naïve theories arising from her "street sense"…
> …
> 7. There should be several levels of formal description, so that a student can move backward and forward from experiential to formal modes of operation.
> ,,,
> 9. It is to be expected that there will be some things you CAN'T do easily in a particular microworld." (Weir, 1987, p.105).

We can see in this characterisation the essential elements of the Object First approach, which we will explore below.

Logo's design was strongly influenced by its MIT stablemate LISP (Berkley, 1964). Logo (McArthur, 1973) is procedural rather than declarative, and has a richer syntax than LISP, making it more suitable for use with beginners. Nonetheless, like LISP, Logo is untyped and uses a list form for all data structures. Like LISP, while Logo is Turing complete, its expressiveness depends heavily on a substantial vocabulary of symbolic operators.

The typical use of Logo for teaching is to start with sequences of concrete turtle operations, introduce fixed repetition, and then explore abstraction through naming sequences as procedures, and generalising operation arguments as variables. Weir notes that:

> "Using variables in this way provides a concrete way of approaching the abstract notion of an algebraic variable as it occurs in school mathematics." (Weir, 1987, p23).

I think that abstraction through the introduction of variables is the key link from coding to programming. However, neither Papert nor Weir offer any guidance for how to do so. We will return to this below.

It is important to remember that Logo was intended to support a constructivist pedagogy that integrated combinatorial thinking and thinking about thinking, not as a programming language per se. Weir seemed to regret the latter use:

> "The books that have appeared since tend to regard "doing Logo" as learning to program. But what about acquiring aesthetic and scientific concepts?" (Weir, 1987, p.11).

Reflecting on his widely read book *Mindstorms*, Papert (1993) acknowledged that:

> "*Mindstorms* unquestionably has a bug for giving prominence to structured programming as a model for thinking about thinking … although *Mindstorms* emphatically proposes the idea of "bricolage" as a model for general scientific theorizing, this idea comes late in the book and is not developed as an alternative style of programming…" (Papert, 1993, p.xv).

There is an enormous literature about Logo and its deployment, much supportive and much critical. There is also an enormous literature about constructivist pedagogy, again much supportive and much critical. I will not explore either here.

Nonetheless, it is salient to reiterate that, in its constructivist conception, Logo was developed to facilitate learners making the transition through bricolage from pre-adolescent concrete thinking to adult abstract thinking. Little consideration was given to its use with adolescents and adults, like senior secondary and undergraduate students, who might be expected to have attained abstract thought, and for whom the bricolage learning style may not be appropriate.

3. From Logo to Objects

From the outset, Logo was widely adopted for school use, at least by schools that could afford the required computing and support infrastructure. For example, an early report for Alberta Education (Kieran, 1984) recommends its deployment across schools, at all levels, for a variety of topics. However, the report is guarded about the use of Logo in Computer Literacy and Computer Science, beyond early years, and recommends avoiding premature use of more advanced features like:

> "procedures, sub-procedures, variables, recursion, or top-down programming" (Kieran, 1984, p.23).

That is, the use of abstraction mechanisms are to be delayed.

Within mainstream Computing education in Higher Education (HE), Logo had considerably less purchase. Programming courses had become universal from the mid-1960s onwards, with content and pedagogy driven largely by changing industrial practices (Michaelson, 2015).

Nonetheless, the motivations behind Logo were taken seriously by both Computing educationalists and language developers. In particular, Logo influenced the evolution of the general purpose object oriented Smalltalk, at Xerox PARC in the early 1970s (Kay, 1993).

Object orientation (OO) long pre-dates Logo: the Simula languages (Dahl, 2004) were developed in the 1960s for discrete event simulation, and Simula 67 was the first widely used OO language. Kay, Smalltalk's designer, acknowledged Siumla as a key influence on Smalltalk (Kay, 1993).

Kay, having encountered Piaget's and Papert's pedagogies through Minsky, visited Papert's team in 1968, and was impressed by Logo's use with children in local schools. His group started teaching Smalltalk to school children from 1973, deploying what were effectively microworlds to underpin learning (Kay, 1993).

Smalltalk is a far more expressive language than Logo, but, like Logo, was implemented from the start in a visual environment. Indeed, one of the first classes constructed in Smalltalk was for Logo turtle graphics (Kay, 1993).

However, Kay observed that non-programmer adults found the transition from very simple to somewhat more complex problems very hard. He analysed a program he thought straightforward to construct, and found 17 "non-obvious ideas":

> "And some of them were like the concept of the arch in building design: very hard to discover, if you don't already know them." (Kay, 1993, p.82).

To address this, Kay's team decided that design should be taught explicitly. His collaborator Goldberg introduced design templates as intermediary forms, to aid decomposing a problem into classes and messages; this approach proved successful.

We could view the use of design templates as a form of *scaffolding*. This constructivist concept was elaborated by Wood, Bruner and Ross (1976), building on Vygotsky's notion of *zones of proximal development* (Vygotsky,1978), in turn developed as a critique of Piaget:

> "*…the zone of proximal development…is the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined by problem solving under adult guidance or in collaboration with more capable peers.*" (italics in original). (Vygotsky, 1978, p.86).

Thus, Wood et al argued that traversing a zone of development requires scaffolding to facilitate it:

> "This scaffolding consists essentially of the adult "controlling" those elements of the task that are initially beyond the learner's capacity, thus permitting him to concentrate upon and complete only those elements that are within his range

> of competence … It may result, eventually, in development of task competence by the learner at a pace that would far outstrip his unassisted efforts." (Wood et al, 1976, p90).

We will return to this notion below.

4. OO and Objects First

Through the 1970s and 80s, mainstream HE Computing continued to be led by industrial imperatives. Impressionistically, while COBOL, and Algol descendants like Pascal, still predominated, we can observe a steady transition to the OO language C++, driven by the rapid uptake of the free, platform independent UNIX operating system from Bell Laboratories.

UNIX was written in C and ran on mini-computers and workstations, offering relatively low cost multi-access, appealing to HE, and a robust software engineering environment, appealing for industrial computing. In turn, C++ was derived from C, and, like Smalltalk, influenced by Simula 67 (Stroustrup, 1987).

In the same period, emerging industrial approaches to OO design, for example Rumbaugh et al's (1990) object modelling technique (OMT), were also adopted in HE. Typically, however, a C++ subset was used to teach traditional procedural programming prior to, and independently of, OO. Smalltalk was not widely adopted.

A fundamental change in undergraduate Computing education came in the mid-1990s, with the widespread adoption of SUN Microsystem's OO language Java (Gosling et al,1996), eventually displacing COBOL for commercial computing, and rapidly gaining use for wider software engineering alongside C++.

Java is disconcertingly like C in appearance, but had OO as the key language design principle from the outset. Unlike Smalltalk and C++, Java supports single rather than multiple inheritance, making it simpler for OO teaching.

Java was complemented by the integration of similar but competing industrial OO methodologies, including OMT, into the Unified Modelling Language (UML) (Booch et al,2005) which rapidly became, and remains, the standard OO design and education practice.

Herein lie the roots of Objects First (OF). Many practitioners noted that students who had already been exposed to procedural programming found the subsequent adoption of OO difficult. It was thought that this might be circumvented by starting with OO. See for example (Wallace and Martin, 1997). Thus, Java and OO with UML quickly displaced procedural programming as the initial undergraduate teaching methodology.

Nonetheless, constructivist thinking, and Papert's pedagogy, were still current in Computing education. For example, Brusilovsky et al (1997) critiqued general purpose languages as being too large, over oriented to numeric and symbolic computation, and lacking visualisation. Citing Logo as a major influence, they advocated the use of mini-languages oriented to constrained domains, much like microworlds, for teaching programming principles, in visual environments. They also advocated the use of subsets of full languages, which they term sub-languages.

Echoing Papert and Weir, Brusilovsky et al commented that:

> "Note that the application of a mini-language is never the goal itself, but a method of mastering a set of notions and skills. If this set contains not only programming concepts, but also some concepts from another domain, a mini-language might be useful to learn this domain (as Logo is used to learn geometry)." (Brusilovsky et al, 1997).

Now, just before the emergence of Java, Kölling, Koch and Rosenberg (1995) critiqued extant OO languages, like C++, Smalltalk and Eiffel, as too complex, and their development environments as too unwieldy, for initial teaching. Subsequently, Kölling and Rosenberg built the Blue OO teaching language (Kölling and Rosenberg,1996a) and development environment (Kölling and Rosenberg,1996b). Serendipitously, Blue shares two key characteristics of Brusilovsky et al's (1997) recommendations: a small language intended to aid learning, implemented in a simple visual environment to support experimentation.

As Kölling acknowledges (2016), academics face an uphill struggle to broaden the use of novel in-house teaching languages, to compete with industrial languages. For example, in 2017, Cass (2017) reported that the top ten languages for IEEE Spectrum readers were Python, C, Java, C++, C#, R, Javascript, PHP, Go and Swift. For comparison, Murphy et al (2017) reported that, in UK Universities in 2016, the top ten teaching languages were Java, Python, C++, C, Javascript, Haskell, C#, Processing, Matlab and PHP.

Thus, the promising outcomes from experiences with Blue were subsequently revisited by Barnes and Kölling (2003), in the BlueJ environment for teaching initial OO in Java.

BlueJ was explicitly developed to support OF (Kölling, 2016, p.13). In their guidelines for teaching with BlueJ, Kölling and Rosenberg (2001) observe that:

> "With BlueJ we can really interact with objects as the very first thing we do. Since objects can be created interactively, the first activity for students should be to open an existing project, create a few objects, make method calls on these objects and inspect the objects' state. Here, we really interact with objects before introducing any other concept. Objects come truly first." (Kölling and Rosenberg, 2001, p.2).

This serendipitously met Brusilovsky et al's (1997) third key recommendation: a microworld like approach to programming education.

In a typical curriculum based on OF, as exemplified say by (Barnes and Kölling, 2003), students are presented with a programming environment for visualising objects, pre-primed with some simple class. Students start by constructing instances of the class, and invoking methods to learn what changes they cause. Next, they start to solve problems that involve modifying instances in specified ways, by sequencing method invocations. Students then explore the code behind the methods, and, with appropriate guidance, modify methods to change their behaviours, and construct new methods with new behaviours.

This approach is far more general than the original microworlds conception. In principle, the scope and properties of the introductory class of objects is limited solely by the instructor's ingenuity and skill, rather than being constrained by a domain oriented notation like the original Logo.

OF in general, and BlueJ specifically, have proved highly popular, and both are still widely used in HE. Nonetheless, the efficacy of OF is controversial.

5. Critiques of Objects First

Hu (2004) provides a thorough survey of limitations to OF. In particular, he observes that OF gives an inadequate grasp of basic algorithms, because arrays are covered late.

Of interest from a constructivist perspective, and contrary to my suggestion above, Hu argues that most young adults lack appropriate abstract thinking capabilities for OO:

> "When addressing the reasons students were unable to see forest in their programming activities, Reek (1995) pointed out that "the problem isn't that the students are stupid, but rather that at age eighteen their thinking maturity is still at the concrete level". Teaching objects-first ignores this fact and thus creating an environment that forces students to think at a higher level of abstraction, which is often the point where the confusion starts." (Hu, 2004, p. 212).

To me, this suggests that OF does not adequately scaffold abstraction. However, Hu recommends a return to procedural programming for beginners, followed by OO.

Lister et al's (2006) comprehensive, but dense, analysis of the SIGCSE mailing list discussion of OF, which considered 99 postings by 39 people, reached no definitive conclusions:

> "There is a fairly strong consensus that programming is hard both to teach and to learn, but the case that objects-early is harder (or easier) than objects-late has not yet been made conclusively." (Lister et al, 2006, p.150).

However, they make the interesting observation that:

> "A key distinction is evident from the debate, however. Two computing sub-disciplines are contending over the role of programming as:
> 1. A manipulative tool for the conduct of algorithmic thought experiments in a purely scientific CS model, as opposed to
> 2. The centrality of design in the construction of large scale software systems by professional software engineers." (Lister et al, 2006, p. 159).

Finally, in their well designed study, Ehlert and Schulte (2009) compared programming learning outcomes for 17 year olds following either an OF or a procedural approach in a first programming course. Both cohorts covered the same topics overall, but in different orders. For OF, students began with instance experimentation, before moving on to variables and control structures. For the procedural approach, students began with variables and control structures, meeting OO much later. The evaluation found that, at the end of the study, the OF cohort had somewhat higher attainment of overall concepts than the procedural cohort, but, in a follow up eight weeks later, there was no significant difference in attainment.

Curiously, both cohorts found arrays and association difficult: the procedural cohort met arrays three topics before the OF cohort, and both met association as the final topic. To me, this suggests that neither approach provided appropriate scaffolding.

6. Computational Thinking, Papert and Objects First

Let us now return to Computational Thinking (CT). In popular discourse, we can discern a spectrum from what we might call weak to strong CT.

At one end, weak CT is toolkit of general purpose, domain independent problem solving techniques that derives from Computing. We have seen that for Papert (1993) problem solving involves "thinking like a computer". Alternatively, CT may be presented as thinking like a computer scientist. For example, for Google (2017):

> The basic skills of computer scientists and the way they think are computational thinking. The area in which you apply CT can be any subject area or topic, even the subject area or topic you teach.

Denning, Tedre and Yongpradit (2017, pp. 32-33) offer a robust, if scattergun, criticism of the alleged primacy and all embracing nature of CT.

Contrariwise, Denning (2017) critiques what he sees as too narrow a conception of CT as problem solving:

> "Underlying all the claims is an assumption that the goal of computational thinking is to solve problems. Is everything we approach with computational thinking a problem? No. We respond to opportunities, threats, conflicts, concerns, desires etc by designing computational methods and tools - but we do not call these responses problem-solutions. It seems overly narrow to claim that computational thinking, which supports the ultimate goal of computational design, is simply a problem solving method." (Denning, 2017, p39).

Between weak and strong CT, Yadav, Stephenson and Hong (2017) list nine core concepts from the Computer Science Teachers Association/International Society for Technology in Education (CSTA/ISTE):

> "data collection, data analysis, data representation, problem decomposition, abstraction, algorithms and procedures, automation, parallelization, and simulation." (Yadav et al, 2017, p. 57).

They also list six CT practices from a College Board/National Science Foundation standalone course:

> "connecting computing, creating computational artefacts, abstracting, analyzing problems and artefacts, communicating, and collaborating." (Yadav et al, 2017, p. 58).

While this offers more precision, nonetheless, on this basis just about any educational activity could be claimed to embody CT.

At the other end of the spectrum, strong CT is a systematic discipline of problem solving. Thus, Kao (2011), following Wing (2006), presents the four key aspects of CT as:

> " …
> - Decomposition: the ability to break down a problem into subproblems.
> - Pattern recognition: the ability to notice similarities, differences, properties, or trends in data.
> - Pattern generalization: the ability to extract unnecessary details and generalize those that are necessary in order to define a concept or idea in general terms.
> - Algorithm design: the ability to build a repeatable, step-by-step process to solve a particular problem." (Kao, 2011,p6).

BBC Bitesize(2017) repeats these almost verbatim, identifying Kao et al's (2011) pattern generalisation as abstraction:

> "There are four key techniques (cornerstones) to computational thinking:
> - decomposition - breaking down a complex problem or system into smaller, more manageable parts
> - pattern recognition – looking for similarities among and within problems
> - abstraction – focusing on the important information only, ignoring irrelevant detail
> - algorithms - developing a step-by-step solution to the problem, or the rules to follow to solve the problem" (BBC Bitesize, 2017)

Note that these aspects lie at the core of the CSTA/ISTE characterisation. Curiously, Google(2017) includes a variant of Kao's definition, while also repeating the CSTA/ISTE list.

Both Papert's approach and OF are based on some pre-given decomposition of a domain, and scaffold algorithm formation through combinatorial thinking, that is by exploring the efficacy of different groupings of rules (Papert) or methods (OF) against some problem requirement.

Here, I think that OF offers little advance on Papert. In particular, there seems to be little principled consideration of what constitute key problem solving and programming concepts, or how their acquisition should be staged. Thus, there are no explicit criteria for choosing an initial object class, or deploying it to scaffold concept acquisition beyond method sequencing.

Furthermore, neither approach:

- scaffolds reflection on the efficacy of potential solutions found by discovery, that is thinking about thinking;
- has anything to say about pattern identification or abstraction;
- offers any guidance on decomposition of a novel problem domain from scratch.

7. Scaffolding programming with patterns and computation structures

While I have critiqued Kao's characterisation as not taking adequate account of information in problem solving (Michaelson, 2015), I am very much of the strong CT persuasion.

I also see problem solving as the heart of programming. However, this view is by no means universal.

For example, Guizdal (2017) suggests that starting with problem solving in teaching Computer Science may be counter productive. Citing Sweller (1988), he asserts that:

> "Problem-solving creates enormous cognitive load that interferes with learning to understand." (Guizdal, 2017, pp. 11).

Instead, Guizdal proposes starting with program comprehension:

> "To teach for understanding, we would give students worked examples and ask them questions about the examples, ask students to predict outcomes or next steps in a visualisation…" (Guizdal, 2017, pp. 11)

While I acknowledge that program comprehension is a vital component of learning to program, I think that problem solving is key to becoming an effective programmer, right from the start. Furthermore, I think that understanding and comparing code fragments is fundamental to abstraction in CT based problem solving.

As I have argued elsewhere (Michaelson, 1992; Michaelson, 2015), I think that problem solving should be driven by abstraction from concrete instances of a specific problem, to identify the constructs appropriate to a general solution. In particular, we should use abstraction to identify the types and variables that an algorithm may manipulate to solve an arbitrary instance.

This approach is in keeping with Vygotsky's (1962) characterisation of the final stages in the development of concept formation in young persons:

> "Only the mastery of abstraction, combined with advanced complex thinking, enables the child to progress to the formation of genuine concepts. A concept emerges only when the abstracted traits are synthesized anew and the resulting abstract synthesis becomes the main instrument of thought. The decisive role in this process, as our experiments have shown, is played by the word,

> deliberately used to direct all the part processes of advanced concept formation.” (Vygotsky, 1962, p78).

That is, as Berger (2005) suggests for mathematics, I see the role of the variable in problem-solving as comparable to the role of the word in Vygotsky’s notion of advanced concept formation.

Here, I think that the CT notion of pattern identification offers a very practical bridge from concrete instances to abstractions with variables. However, the term “pattern” may be misleading. We normally understand a pattern to have some repetitive fixed structure, for example in tiling a floor as a black and white chequerboard. But, for problem solving, we actually want to identify *the underlying regularities in differences*. So a pattern is *some property that lots of different instances share*; ideally one that enables us to *generate new instances* that also share the property.

I also think that the notion of identifying a pattern should be more than just a metaphor for some informed intuition that comes with experience. Rather, pattern identification should be taken literally, as an approach that involves directly comparing structural and operational features in solving concrete instances of problems, to identify how they are similar and how they differ.

This generalises Papert’s combinatorial thinking, where the learner behaves within a constrained domain to write down sequences of actions for the computer to perform. In an arbitrary domain, we proceed by solving lots of concrete instances of a problem, analysing what we did with the concrete data to get to the concrete results, and then looking for patterns across instances in both data and actions. We can do this by decomposing our data and actions at finer and finer levels of detail until we can identify and generalise the patterns amongst and across elementary operations on elementary data. This is reminiscent of Turing’s analysis of someone doing arithmetic with pencil and squared paper, going to right down to symbol by symbol operations (Turing, 1936).

Furthermore, I think we should view patterns as templates, encompassing both information and computation:

- patterns in data lead to variables and expressions to calculate with them;
- patterns in computations lead to assignments, structured programming constructs and subprograms.

Thus, I envisage patterns as skeletal forms in some well defined notation, for example a pseudo code or reference language (Scottish Qualifications Authority, 2015), or, indeed, some programming language. That is, ultimately, patterns are rendered as chunks of syntax with holes in them.

Note that, here, there are fundamental differences to OO design patterns (Gamma et al, 1995) which are used to structure complex programs constructed from classes with well defined interfaces. This usually involves retrofitting extant components to a pattern. In my conception, control patterns have closer analogies with higher order function from functional programming (Michaelson, 1992). These are used to abstract common patterns of recursion for instantiation with context specific functions to control repetition and computation.

7.1. Finding variables and expressions

An elementary expression typically has variables for values that change and constants for values that don't change, combined by unchanging operations. We can find expressions by abstracting over concrete calculations.

For example, suppose we want to travel 400 kilometres. If our vehicle goes at 100 kilometres per hour, the journey takes:

400/100 == 4 hours.

And if our vehicle goes at 80 kilometres per hour, the journey takes:

400/80 == 5 hours.

Comparing the expressions, they have “400/” in common so we can abstract where they differ:

400/? hours.

Now, we can introduce a variable called “speed”, which has to be integer like 100 and 80, and then for unknown values find:

400/speed hours

We all know how to do this. How about making it explicit?

7.2. Finding computation patterns

Let's next consider three archetypal computation structures and their corresponding programming language forms: choice, repetition and iteration.

First of all, an *if* statement typically has a choice condition, with branches depending on whether it is true or false. We can view this as a pattern for discriminated computation. In turn, this may depend on being able to identify a pattern to divide data into two groups, where each group is processed in the same manner. That pattern then frames the condition.

Next, a *while* statement typically has a termination condition and a body. We can view this as a pattern that determines when to stop performing a repeated computation. In turn, this may depend on being able to identify a pattern in either data or actions, characterising how each step determines the next.

Finally, a *for loop* typically has a control variable with initialisation and incrementation expressions, a termination condition, and a body. We can view this as a pattern for counted or indexed computation. In turn, this may depend on being able to identify a pattern characterising a sequence of data, where each element is processed or generated in the same manner, or processing each element depends on some property of the previous element.

In all three cases, our understanding of the computation pattern guides our abstractions. That is, when we explore data looking for computation patterns, we should ask of it questions that enable us to choose amongst the possibilities.

All three computation patterns involve:

- a condition that determines what to do next;
- what is done next;
- what it is done to.

So maybe we might interrogate our data by asking:

- what have we got?
- what are we doing to it?
- why or when are we doing it?

7.3. Example

Suppose we're organising a sports club outing to the chariot racing, and we want to work out how many full price and how many concession price tickets we need to purchase. Suppose we know everyone's age:

12 44 23 63 13 69 10 12 61 …

and that anyone under 16 or over 60 can get a concession.

To begin with, let's write down what we do with at each concrete age in turn:

data	*action*	*condition*
12	count 1 concession	under 16
44	count 1 full price	between 16 and 60
23	count 2 full price	between 16 and 60
63	count 2 concession	over 60
13	count 3 concession	under 16
69	count 4 concession	over 60
…	…	…

We've already implicitly abstracted out accumulation variables for the concession and full price counts, just by describing what we're doing.

On closer inspection, we can see that the action for each concrete data item involves incrementing a count depending on which condition is met:

data	*action*	*condition*
12	increment concession	under 16
44	increment full price	between 16 and 60
23	increment full price	between 16 and 60
63	increment concession	over 60
13	increment concession	under 16
69	increment concession	over 60

…	…	…

We can regroup by the three condition:

data	*action*	*condition*
12	increment concession	under 16
13	increment concession	under 16
44	increment full price	between 16 and 60
23	increment full price	between 16 and 60
63	increment concession	over 60
69	increment concession	over 60
…	…	…

Now we can spot the choice structure pattern, abstract over the age in the three conditions, and introduce an if statement for each:

```
IF age under 16 THEN
 increment concession
ELSE
 IF age over 60 THEN
  increment concession
 ELSE
   increment full
 END IF
END IF
```

Alternatively, we can look for a pattern based on grouping abstracted actions:

data	*action*	*condition*
12	increment concession	under 16
13	increment concession	under 16
63	increment concession	over 60
69	increment concession	over 60
44	increment full price	between 16 and 60
23	increment full price	between 16 and 60
…	…	…

Again, we can abstract over the age but introduce two if statements having combined conditions for common actions:

```
IF age under 16 OR over 60 THEN
 increment concession
ELSE
  increment full price
END IF
```

We can now think about how processing the sequence of data, rather than each data item, is structured. Let’s make the process of dealing with each item in turn more explicit:

data	*action*	*condition*
	look for more data	more data
12	…	…
	look for more data	more data
13	…	…
	look for more data	more data
63	…	…
…	…	…
	look for more data	no more data
	stop	

This looks like a repetition structure of the form:

```
look for more data
WHILE more data DO
  IF … THEN … ELSE …
   look for more data
END WHILE
```

Again, we all know how to do this sort of blow by blow analysis and, with experience, we come to do so in big "intuitive" steps. But beginners don't know where to start. Again, how about making this analysis explicit?

7.4. Finding information structures

Note that the notion of a type encompasses structured information, for example linear sequences and nested structures, not just base types like numbers. However, identifying how information is structured need not lead us immediately to storing data in such a structure. That depends on whether processing the information requires it to be stored for subsequent access.

For example, it is tempting to automatically identify a linear sequence of numbers with an array. However, finding the average of a sequence of numbers can be performed by taking each number in turn. In contrast, finding all the numbers less than the average requires them all to be held until the average is known. Again, this is something we can tell by systematic and detailed working with concrete instances.

8. Conclusions

Teaching programming is hard. It seems deeply unsatisfactory that some people seem to get it and some don't, and that we don't understand why.

After critiquing the bricolage/microworld and Objects First approaches to teaching programming, I've argued that we might better deploy a pedagogy based on strong CT. That is, I think that we can teach beginners how to systematically analysis concrete instances of problems, to tease out the essential patterns in data and computations, leading to abstractions that can form the basis of algorithms and, ultimately, programs.

Acknowledgements

I would like to thank Nancy Falchikov for valuable discussions about constructivism, development and cognition.

References

D. Barnes and M. Kölling (2003), *Objects First with Java: A Practical Introduction using BlueJ*, Prentice-Hall.

Berger, M. (2005). Vygotsky's theory of concept formation and mathematics education. In H. L. Chick & J. L. Vincent (Eds.), *Proceedings of the 29th conference of the International Group for the Psychology of Mathematics Education* (Vol. 2, pp. 153–160). Melbourne: PME.

BBC Bitesize (2017), *Introduction to computational thinking*, http://www.bbc.co.uk/education/guides/zp92mp3/revision (inspected 16/5/17)

E. C. Berkley and D. G. Bobrow (eds) (1964). T*he Programming Language LISP: Its Operation and Applications*, MIT Press.

G. Booch, J. Rumbaugh and I. Jacobson (2005). *The Unified Modeling Language User Guide, 2nd Edition*, Addison-Wesley.

P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko and P. Miller (1997). Mini-languages: A Way to Learn Programming Principles. *Education and Information Technologies* 2(1), pp. 65-83.

S. Cass (2017), The 2017 Top Programming Languages, *IEEE Spectrum*, 18th July, 2017.

O. J. Dahl (2004). The birth of object orientation: the Simula languages, in *From Object-Orientation to Formal Methods*, Springer.

P. J. Denning (2017). Remaining Trouble Spots with Computational Thinking, *CACM*, 60(6), pp. 33-39.

P. J. Denning, M. Tedre and P. Yongpradit (2017). Misconceptions About Computer Science, *CACM*, 60(3), pp. 31-33.

A. Ehlert and C. Schulte (2009). Empirical Comparison of Objects-First and Objects-Later, *Proceedings of the Fifth International Workshop on Computing Education Research* (ICER'09), ACM pp. 15-26.

Google (2017). What is Computational Thinking? in *Computational Thinking for Educators*, https://computationalthinkingcourse.withgoogle.com/unit?lesson=8&unit=1 (inspected 16/5/17)

E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reuseable Object-Oriented Software*, Addison-Wesley.

J. Gosling, B. Joy and G. L. Steele Jr. (1996). T*he Java Language Specification*, Addison Wesley.

M. Guizdal (2017), Balancing Teaching CS Efficiently with Motivating Students, *CACM*, 60(6), pp10-11.

C. Hu (2004). Rethinking of Teaching Objects First, *Education and Information Technologies*, 9(3), pp. 209–218.

A. C. Kay (1993). The Early History of Smalltalk, *ACM SIGPLAN Notices*, 28(3), pp.69-95.

E. Kao (2011). Exploring Computational Thinking at Google, *CSTA Voice*, 7(2), p.6.

T. E. Kieren (1984). *LOGO in Education: What, How, Where, Why and Consequences*, Planning Services, Alberta Education.

M. Kölling, B.Koch, and J. Rosenberg (1995). Requirements for a First Year Object-Oriented Teaching Language, *ACM SIGCSE Bulletin,* 27(1), pp. 173-177.

M. Kölling and J. Rosenberg (1996a). Blue – A Language for Teaching Object Oriented Programming, *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, U.S.A., SIGCSE Bulletin 28(1), pp 190-194.

M. Kölling and J. Rosenberg (1996a). An Object Oriented Development Environment for the First programming Course, *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, U.S.A., SIGCSE Bulletin 28(1), pp 83-87.

M. Kölling and J. Rosenberg (2001). *The Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001),* Canterbury.

M. Kölling (2016). Lessons from the Design of Three Educational Programming Environments: Blue, BlueJ and Greenfoot. *International Journal of People-Oriented Programming*, 4 (1). pp.
5-32.

R. Lister, A. Berglund, C. Clear et al (2006). Research Perspectives on the Objects-Early Debate, *Working group reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '06),* ACM, pp. 146-165.

C. D. McArthur (1973). *LOGO User's Guide and Reference Manual*, Bionics Research Reports: No. 14, Bionics Research Laboratory, School Of Artificial Intelligence, University of Edinburgh.

G. Michaelson (1992), *Elementary Standard ML*, UCL Press.

G. Michaelson (2015), Teaching programming with computational and informational thinking, *Journal of Pedagogic Development*, 5 (1), pp. 51-65.

M. Minsky and S. Papert (1971). *Artificial Intelligence Research Report*, Memo AIM-252, AI Laboratory, MIT.

L. Moors and R. Sheenan (2017), Aiding the Transition from Novice to Traditional Programming Environments, Proceedings of IDC 2017: ACM Interaction Design and Children Conference, Stanford University, pp509-514.

E. Murphy, T. Crick and J. H. Davenport (2017), An Analysis of Introductory Programming Courses at UK Universities, *The Art, Science, and Engineering of Programming*, Vol. 1, No. 2, 2017, Article 18; 23 pages.

S. Papert (1993), *Mindstorms* (2nd Edition), Basic Books.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1990). *Object-Oriented Modeling and Design*. Prentice Hall

M. Reek (1995). A top-down approach to teach programming. *SIGSCE Bulletin*, 27(1), pp. 6–9.

Scottish Qualifications Authority (2015). *Reference language for Advanced Higher Computing Science question papers*, http://www.sqa.org.uk/files_ccc/ComputingScienceReflanguageSpecificationsSQPAH.pdf (consulted 17/5/17).

B. Stroustrup (1987). *The C++ Programming Language*, Addison-Wesley.

J. Sweller (1988), Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12, pp. 257–285.

M. Tedre and P. Denning (2016). The Long Quest for Computational Thinking. *Proceedings of the 16th Koli Calling Conference on Computing Education Research* , November 24-27, 2016, Koli, Finland, pp. 120-129.

A.Turing (1936). On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, 42 (1), pp. 230-265.

L. S. Vygotsky (1962). *Thought and Language*, MIT Press.

L. S. Vygotsky(1978). *Mind in Society*, MIT Press.

J.M. Wing (2006). Computational Thinking, *CACM Viewpoint*, March, 33-35.

C. Wallace and P. Martin (1997). Not **Whether** Java but **How** Java, *Proceedings of Java in the Computing Curriculum Conference (JICC 1)*, South Bank University

S. Weir (1987). *Cultivating Minds: A Logo Casebook*, Harper and Row.

D. Wood, J. S. Bruner and G. Ross (1976). The Role of Tutoring in problem Solving, *Journal of Child Psychology and Psychiatry*, 17, 89 -100.

A. Yadav, C. Stephenson and H. Hong (2017) , Computational Thinking for Teacher Education, *CACM*, 60(4), pp. 55-62.