Invited Paper, 17th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ), Christchurch, New Zealand, July 2004, pp. 13-19.

Objectives and Object-Oriented Programming

Dr. Raymond Lister
Faculty of Information Technology,
University of Technology, Sydney
NSW, Australia
Email: raymond@it.uts.edu.au

After decades of experience, the computer education community has settled upon a relatively standard way of teaching procedural programming. With the growing use of object-oriented languages, the question is whether new ways are required. Some in the community argue that object-oriented programming languages are an extension of the 3GL approach, and students should still be taught the 3GL way first, even if students are using an object-oriented language. On the other hand, the "objects early" proponents argue for radical change. As with most debates, the points raised by both sides contain implicit assumptions. Our real differences of opinion are at a more abstract level: our teaching objectives. This paper describes my experiences in trying to make teaching objectives more explicit, within the domain of object-oriented programming. First, I discuss PigWorld, a micro-world for teaching programming that blends "objects-early" with a traditional emphasis on algorithms. Second, I describe my use of Bloom's taxonomy to make my assessment objectives more explicit. Finally, I look beyond introductory programming, to describe how an explicit acknowledgement of objectives clarifies the debate on whether the teaching of data structures needs to change if we teach "objects early".

Keywords:

CS1, objects-early, Bloom's taxonomy, CS2, Data Structures, phenomenography.

1. INTRODUCTION

Computing educators are not lighthouse keepers: we do not work alone. We work within several communities: from a world-wide community, national communities, down to our own faculty or department. Within all these communities, there are specifications on many educational issues. At an international level, organizations like the Association of Computing Machinery publish guidelines on curricula. At a national level, organizations like the NACCQ publish more specific frameworks. At the faculty or department level, there are frequently quite detailed specifications for what shall be taught, how it is to be taught, and how it is to be assessed.

Despite all these specifications, there is still plenty of argument about how computing should be taught. There may be more contested territory in computing than in other disciplines, given the rate of change of information technology. Some see the Internet and object-oriented programming as new developments that require changes to our teaching, while others argue that the curriculum should emphasize the fundamental concepts that do not change.

Most computing education discussion at the faculty/department level is of a poor standard. Discussion is usually about a very concrete issue, such as which programming language should be taught first, when the real difference of opinion is usually at a higher level of abstraction, and remains implicit. In their popular book on negotiation, Fisher, Ury, and Patton (1991) describe a lower-level issue as a "position", and a higher level abstraction as an "interest". They write, "Interests motivate people; they are the silent movers behind the hubbub of positions. Your position is something you have decided upon. Your interests are what caused you to decide" [p41]. For example, advocating that Java be taught as the first language is a position. Your interest in doing so could be that you want a language that (you hope) will inspire students to study harder.

When debating education issues, we usually see it as a monotonically decreasing process: we either eliminate positions until there is a single "winner", or we merge positions by negotiating a compromise. Fisher, Ury, and Patton argue that we should actively seek to invent new options that satisfy most interests of most parties. For example, having established that your interest is inspiring students to study harder, can that be achieved without changing to a particular programming language? Alternately, having established that your interest is the teaching of fundamental concepts that do not change, perhaps the introduction of a new programming language is the ideal opportunity to demonstrate that "the more things change, the more they stay the same".

Currently, object-oriented programming generates most discussion on curriculum change. In the remainder of this paper, I shall describe three experiences of my own, where I have sought to invent options to accommodate multiple interests while introducing Java into the undergraduate curriculum.

2. ALGORITHMS VERSUS OBJECTS EARLY

Some computer scientists argue that the teaching of the procedural programming paradigm must precede the teaching of object-oriented programming. Burton and Bruhn (2003) argued as follows:

> "*... while OOP undeniably represents a new paradigm, it in no way replaces the older paradigm ... rather it is in addition to it. As a paradigm in its own right algorithmic thinking (together with procedural programming) needs to be understood first, and in isolation from OOP. Students need to know how OOP fits into the bigger picture and a first course is surely required to do this. ... all students should be given a firm foundation in algorithms, structured programming, procedures, and an appreciation of this historical development, before proceeding to object-oriented programming.*"

The above is a commonly expressed view, which Culwin (1999) breaks it into two related strands:

> "*The first is that the new paradigm is more complex and hence more difficult than the old paradigm. The second is that knowledge and experience of the old paradigm is a necessary pre-requisite for working effectively within the new paradigm. The first of these justifications is possibly true but the second is based upon an obvious fallacy. The people who decide upon the nature, content and focus of undergraduate curriculum are, in general, of an age where their own professional and intellectual development mirrors the development of computing over the last twenty years or so. Hence their personal perception of how they learned ...* [is generalized] *... to a conception that knowledge of the old paradigm is a necessary pre-requisite for learning the new paradigm...*"

2.1 PigWorld

PigWorld (Lister, 2004) demonstrates that there is no need to choose between objects-early and algorithms-early. The apparent need to do so is caused by a widespread misunderstanding of what it means to encode algorithms in the object-oriented style. The misunderstanding is prevalent even among those who advocate objects-early. The algorithm-object dialectic is transcended by the following general principle:

> *In the procedural style, algorithms are encoded explicitly within the method of an object, but in the object oriented style, algorithms emerge implicitly from the interactions between objects.*

This general principle is illustrated in the remainder of this section, by describing how pigs in PigWorld use "flower power" to traverse a maze. Figure 1 shows a simple PigWorld scenario, with one pig. There are also pig food trees, which periodically drop pig food pills onto squares adjacent to the tree. Hungry pigs move, one square at a time, toward the nearest pig food. As a pig moves, it leaves a trail of flowers.

When a pig has eaten sufficient food, the pig will become "in the mood for love". Boy pigs in such a mood move to the nearest girl pig. If the girl pig reciprocates the boy's amour, they mate to produce a new pig. When first demonstrating PigWorld to a new undergraduate class, the creation of the first baby pig is met with laughter and applause: sex sells, especially in an undergraduate class.

Robots (or creatures) that traverse mazes have been a staple of teaching introductory programming for over 20 years (see Lister, 2004, for citations). However, those earlier approaches have traversed mazes in the procedural style, with the robots/creatures maintaining internal data structures to map their path through the world. Pigs traverse the maze without such data structures, in the object-oriented style, by interacting with flowers. A flower has only one real encapsulated property, its age, which increments with each time step in PigWorld. If a pig moves on to a square that already has a flower, then the old flower is replaced with a new flower of zero age. When a pig chooses the next square to which it will move, it does so according to the following three criteria: (1) A square not containing a flower is preferred; (2) If there is a choice between moving to two squares, neither of which contains a flower, then the square in the direction nearest to the pig's intended target (e.g. food) is preferred, and (3) if all adjacent squares contain flowers, then the square with the oldest flower is preferred. Thus a pig's maze traversal proceeds according to the principle enunciated earlier: algorithms emerge implicitly from the interactions between objects.

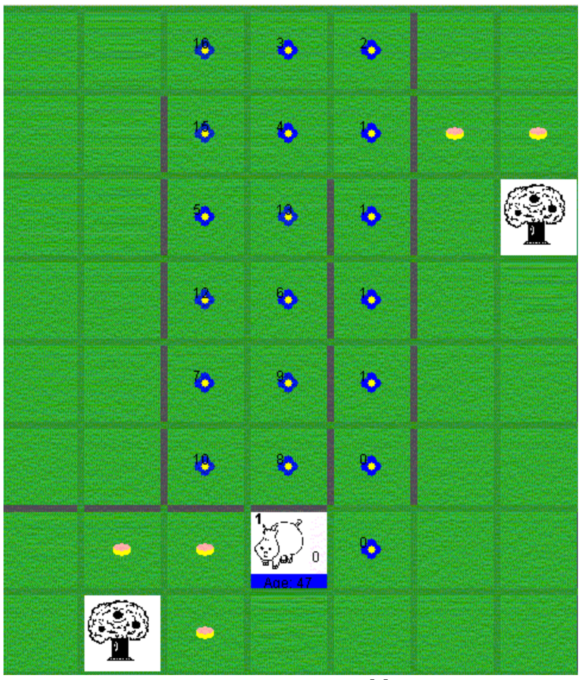
Figure 1. PigWorld

Even a mathematically rigorous introduction to algorithms can be done in the object-oriented style. For those who doubt that assertion, consider this problem: either prove that a single pig in any connected maze will eventually visit all squares using "flower power", or provide a counter example.

2.2 Linked Lists Early

In the philosophy of procedural programming, one cannot realistically talk about algorithms without also talking about data structures. In teaching procedural languages, the first data structure taught is the array. In teaching objects-early, the first data structure taught should be the linked list. Algorithms for manipulating linked lists are more complicated in the procedural style than in the object-oriented style. The recursive code of procedural programming is elegant, but too difficult for the novice. On the other hand, implementing a linked list in Java requires no more than a grasp of the most basic concepts (objects, message passing, parameters) plus the "if" statement – what more natural way to introduce these concepts (and make clear the distinction between classes and objects) than via the linked list? The message passing nature of object-oriented programming code simplifies many issues of control flow, and provides much of the same style and elegance of recursive code.

We propose the following principle about data structures, which is a special case of the earlier principle on the algorithm-object dialectic:

> *In the procedural style, data structures are controlled by algorithms outside the data structure, but in the object oriented style, algorithms emerge from the interactions between the objects containing the data.*

We now illustrate this principle by describing recent experiences with PigWorld (Lister, 2004). In a recent semester, students were required to modify PigWorld so that pigs traversed the maze by trailing a rope, rather than dropping flowers. Once a pig satisfied its current goal (e.g. located food), the pig was supposed to "roll up" its rope and begin a fresh maze traversal. An instance of the Rope class is associated with a square. Each instance has the references of up to two other pieces of rope, the instance laid out by the pig before this instance, and the instance laid out after. Thus, the entire trail of rope is a doubly linked list. The code that allowed pigs to lay out rope was supplied to students, and students were required to write code to "roll up" the rope, implemented as the destruction of the linked list. Each instance in the linked list need only pass along a "destroy" message, which requires less than 10 lines of code. Of the students who attempted this assignment task, 82% were successful (Lister, 2004).

In the most recent semester, a new type of creature was added to PigWorld, the snake, that may occupy several squares simultaneously. The snake's body was implemented as a linked list, and students were required to write code that passed messages along the body of the snake. In a reflective document written as part of the assignment, one student wrote, "I have messed around with programming languages for several years and was always intimidated by linked lists. Here they are pretty harmless actually."

If we accept that linked lists are an easy and natural data structure to teach "objects early", then it becomes obvious that there need be no tension between those who advocate "objects early" and those who advocate an early emphasis on algorithms.

3. MAINTAINING STANDARDS VERSUS DUMBING DOWN

Irrespective of whether we teach the object or procedural styles, the sad fact is that many students struggle with programming. The results from a multinational project led by McCracken (2001) indicate that the problem is world-wide, and transcends the 3GL and object-oriented paradigms. In the McCracken study, the ten authors in four countries tested their respective students on a common set of programming tasks. Most students did not even get close to solving the tasks. McCracken *et al.* noted that many weak students became so engrossed in overcoming syntax errors that they lost sight of the problem to be solved, and when they achieved their first clean compile, were "surprised" by what the program did when presented with data.

Most people who have taught introductory programming will relate to the McCracken experience. Many of us have invested enormous amounts of time to structure lecture materials for weaker students, and provide helpful lab exercises. When we succeed, and the failure rate for introductory programming falls, our reward is to be accused by our colleagues of having "dumbed down" introductory programming. It is as if we face two incompatible constituencies: our students and those who teach "downstream" from us.

The dialectic articulated in the heading of this section appears most difficult to transcend. Only those of us who have direct contact with novice programmers develop a real appreciation of just how difficult programming is for many novice students, but only those who teach further downstream develop a real appreciation of what our graduates can really do. There even appears to be a moral dimension to the dialectic: some tertiary teachers see their role as filtering out "the dross", while others see their role as helping the students who struggle.

3.1 Bloom's Taxonomy

Bloom's taxonomy (Bloom, 1956) can clarify any debate on "maintaining standards" versus "dumbing down". The taxonomy contains six levels, which from lowest to highest are: Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation. Each level of the taxonomy describes a type of competence in some domain. The key principle of Bloom's taxonomy, the principle that clarifies the debate on "maintaining standards" versus "dumbing down", is that competence at a higher level of the taxonomy implies a reasonable degree of competence at the lower levels.

When competent at the knowledge level, a student can regurgitate a fact when prompted for it, without necessarily understanding the significance of the fact. Students can operate at the knowledge level merely by rote learning. That is not a level of competence of much interest in tertiary education, but it serves a purpose much like that of the base case of a recursive function: it identifies the trivial case.

In absence of any knowledge of this taxonomy, debate about teaching frequently confuses the concepts represented by the knowledge level and the next level up, the comprehension level. This level is a higher level of competence than the knowledge level because a student competent at the comprehension level understands the significance of a fact. A student can demonstrate competence of programming at the comprehension level in several ways. One way is to translate a piece of novel pseudo code into working code – an excellent lab exercise and practical exam Another way is to correctly predict the output of a novel piece of code, which lends itself well to assessment by multiple choice questions. These tasks describe a competence significantly less than the ability to actually write code, but the philosophy behind Bloom's taxonomy is that we should assess students at this comprehension level before we assess their capacity to write code.

The lesson of the McCracken project was that many students around the world currently receive passing grades in programming subjects, despite not being able to program. I believe the problem has a two-fold cause. The first part of the cause is the unhealthy concentration on assessing students by requiring them to write code. (What is the point in doing that when, as McCracken *et al.* observed, the weak students are "surprised" by what their code does after the first clean compile!) The second part of the cause is that, having set students a programming task, we frequently give half marks – a pass – to students who submit poorly structured, buggy programs. The true measure of a teacher's standards is not to be found in their syllabus outline, their lecture notes, nor even their exam. The true measure of a teacher's standards is to be found in how they mark. Teachers who set difficult programming assignments are not maintaining standards if they hand out passing marks for bad solutions. Indeed, they may be guilty of covert dumbing down.

At my university, programming in the first and second semester is now assessed explicitly according to Bloom's taxonomy (Lister and Leaney, 2003a, 2003b). To pass the first semester, a student need only demonstrate competence at the comprehension level.

(The students are expected to demonstrate higher competencies in subsequent semesters.) In that first semester, students seeking the lowest possible passing grade are assessed in two ways. First, in a practical exam, they are required to translate a piece of novel pseudo code into working code. Second, they must achieve 70% or higher in a multiple choice exam, where questions typically ask them to predict the output of a piece of code, or nominate the missing line from a short piece of code. We find the multiple choice exam to be a more demanding test than requiring students to write code, as there is only a single correct answer to these multiple choice exams, whereas when students provide written answers, it is all too easy to conceal our teaching mistakes by marking generously.

My interest in teaching programming is no different from any other teacher: I want students to be competent – eventually - at the highest levels of Bloom's taxonomy. The difference in my position is merely a matter of timing: I do not expect the weakest passing students to be competent at designing programs until their third semester. In that third semester, our students are taught the classic, 3GL data structures, in the "C" programming language, with a heavy emphasis on students writing programs. The third semester failure rate has a long history of being very bad. Since our change in the first year to a grading approach explicitly based on Bloom's taxonomy, the failure rate of that third semester subject has plummeted, and within the department this improvement is commonly attributed to the change in how students are assessed in their first year.

Teaching a lot of material and then assessing it badly is not "maintaining standards". We should all aim to teach less, but assess it thoroughly: that is not "dumbing down".

4. TEMPLATE LIBRARIES VERSUS IMPLEMENTING DATA STRUCTURES

If novice programmers are being taught objects-early, should changes now flow through to the rest of the programming curriculum? For example, should the teaching of data structures place less emphasis on the direct implementation of data structures and more emphasis on the good use of existing data structures in the template libraries of C++ and Java?

Debate on the teaching of data structures illustrates beautifully the need to debate interests rather than positions. A typical staff room debate on data structures will revolve around a position, such as "the students must implement a balanced binary tree", and not the higher level skills we hope the students will gain from that exercise.

4.1 The Dimensions of Variation in the Teaching of Data Structures

Recently, I was part of a project to determine the interests behind the teaching of data structures (Lister et al., 2004). We used a research method known as phenomenography. It is not necessary to give a detailed description of that method here. Suffice to say that we collected data by interviewing some teachers, and we also collected position statements from the introductions of some text books. We then analyzed that data to identify the interests that were behind the stated positions. We identified five interests that teachers have in teaching data structures:

1. Developing Transferable Thinking: Data structures are a vehicle for developing thinking skills that have relevance beyond their immediate application to data structures. Just as PigWorld provides a micro world in which students can learn object-oriented concepts, data structures provides a micro world in which students develop the thinking skills that transfer to other programming tasks.

2. Improving Students' Programming Skills: Especially student dexterity with recursion and pointers. As one interviewee expressed it, "reading and using the code without having written something similar is like watching Olympic ping pong on TV. It sure looks easy, even somewhat repetitious; however, the level of precision is only experienced by trying to do the same."

3. Knowledge of Software Libraries: This category gives a central role to teaching data structure libraries. Of all the views, it is the most utilitarian, seeing data structures as a set of tools used for solving problems. As one interviewee expressed it, "... many career paths will never lead the graduate to read or write code which implements the operations of a binary search tree, B-tree, hash table, heap-structured priority queue, etc. So for these structures, it's enough to know how to read and write code that uses them, based on their presence in good collection libraries."

4. Component Thinking: This category sees object-oriented programming as having a design methodology beyond that of procedural programming, emphasizing code reuse, abstraction, information hiding, and patterns.

5. Knowing "What's Under the Hood": The assumption is that before students can make effective use of the data structures in the template libraries of C++ Java, students must develop insight into how those data structures are implemented.

It is important to understand that an individual teacher does not typically subscribe to just one of the above interests. There may be interests to which an individual identifies strongly, other interests to which they identify weakly, and some interests with which they do not identify at all.

From the first four of the above five categories, we identify two dimensions in which the categories differ. These dimensions are shown in Table 1. In one of the dimensions, the variation is in the degree of abstraction. The categories "Improving Students' Programming Skills" and "Knowledge of Software Libraries" both emphasize implementation skills, whereas the categories "Developing Transferable Thinking" and "Component Thinking" both emphasize the design process. The other dimension of variation is "Computer Science" versus "Object Engineering". The category "Developing Transferable Thinking" relates to the Turing Machine as a universal computational device, while the category "Improving Students' Programming Skills" relates to the realization of the Turing Machine in the von Neumann architecture. On the other hand, "Object Engineering" is not about building universal computational devices, but instead devices that are well suited to specific purposes. The fifth category "Knowing What's Under the Hood" transcends the Computer Science vs. Object Engineering dialectic, but it is more concrete than abstract.

	Computer Science vs. Object Engineering	
Abstract vs. Concrete	Developing Transferable Thinking	Component Thinking
	Improving Students' Programming Skills	Knowledge of Software Libraries

Table 1. The Dimensions of Variation in the Teaching of Data Structures

Having identified the real differences behind various positions, we can now reinterpret the staff room debate: "that students must implement a balanced binary tree". Teachers who subscribe strongly to either or both categories in the "Computer Science" dimension might agree, because the exercise will strengthen a student's skills (although each of those two categories emphasises different skills). However, a teacher who subscribes strongly to either or both categories in the "Object Engineering" dimension is likely to disagree. A teacher who subscribes to the fifth category "Knowing What's Under the Hood" might agree, but some teachers who subscribe to this category might also disagree, believing that it is sufficient for students to understand the concept of a balanced binary tree, and be aware of its time complexities.

This phenomenographic study demonstrates a more constructive approach to syllabus design. Most discussion on syllabus design is done face-to-face, in an adversarial style. Many teachers avoid such confrontation, or do not perform well in such a "hostile" environment, despite having something of value to contribute. This may explain a recent finding that the primary driver of curriculum change is influential or outspoken individuals, with pedagogic arguments being a very minor driver of change (Gruba et al., 2004). If a syllabus review process began with a phenomenographic study, more teachers might articulate their position. In this phenomenographic study, my own view changed as the study progressed. One of the insights for me was the distinction that emerged between "Component Thinking" and "Knowledge of Software Libraries". Prior to the study, I had been inclined to think they were the same category, and I believe that is a common mistake in our community. People who advocate a greater emphasis on the template libraries of C++ and Java are frequently and incorrectly shouted down as advocating nothing but the teaching of an application program interface.

5. CONCLUSION

In our teaching communities, we are prone to foster a culture that focuses upon our internal differences, not what we have in common. We may differ on what programming language should be taught first, but we all do want our students to program. We may differ on what a student should be able to do after one semester, but there is a high level of agreement on what they should be able to do by the time they graduate. We may disagree about how far down into the bowels of the von Neumann machine all students must go, but we do agree that the primary reason for sending them there is to make them better software engineers. Behind our many different, detailed positions, we have remarkably similar interests: that is why we are a community.

When we disagree, we should go back to first principles, and re-establish what we agree upon. We might then be surprised that apparent differences are really just implementation details that are amenable to solution - provided we are willing to negotiate, willing to change, and provided we approach our teaching with imagination.

REFERENCES

Bloom, B.S. (Ed.) (1956) "Taxonomy of Educational Objectives: Handbook I: Cognitive Domain", Longmans, Green and Company.

Burton, P.J., Bruhn, R.E. (2003) "Teaching Programming in the OOP Era". SIGCSE Bulletin, 35:2 (June), pp 111-114.

Culwin, F. (1999), "Object Imperatives!"Proceedings of the 30th Technical Symposium on Computer Science Education (SIGCSE), New Orleans LA, USA, March, pp 31-36.

Fisher, R., Ury, W., and Patton, B. (1991). "Getting To Yes: Negotiating Agreement Without Giving In". New York, NY: Penguin. 2nd edition.

Gruba, P., Moffat, A, Sondergaard, H., Zobel, J. (2004) "What Drives Curriculum Change?" Proceedings of the Sixth Australasian Computing Education Conference (ACE). Dunedin, New Zealand, 18-22 January, pp 109-117.

Lister, R. and Leaney, J. (2003a) "First Year Programming: Let All the Flowers Bloom". Proceedings of the Fifth Australasian Computing Education Conference (ACE). Adelaide, Australia, 4-7 February, pp 221-230.

Lister, R and Leaney, J (2003b), "Introductory Programming, Criterion Referencing, and Bloom". Proceedings of the 34th Technical Symposium on Computer Science Education (SIGCSE), Reno, Nevada USA, 19-23 February, pp 143-147.

Lister, R. (2004). "Teaching Java First: Experiments with a Pigs-Early Pedagogy". Proceedings of the Sixth Australasian Computing Education Conference (ACE). Dunedin, New Zealand, 18-22 January, pp 177-183 .

Lister, R., Box, I., Morrison, B., Tenenberg, J., Westbrook, S. (2004) "The Dimensions of Variation in the Teaching of Data Structures". Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE), Leeds, UK, 28-30 June.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001) "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students", SIGCSE Bulletin, 33:4, (December), pp 1-16.