Teaching Java Backwards

Philip Machanick*
School of ITEE
University of Queensland
St Lucia, Qld 4072
Australia
philip@itee.uq.edu.au

Abstract

Presenting material in a more object-oriented way is a growing trend in Computer Science education. This paper takes the idea of objects-first or abstraction-first teaching a step further, by using Bloom's Taxonomy to design a course in such a way as to present factual content early, followed by higher-level cognitive skills. In the course described here, factual content was covered rapidly, then reinforced by assignments, laboratory sessions and tutorials, aimed at building higher cognitive skills. The resulting course was successful in relatively rapidly bringing a class doing a "bridging" diploma up to the level required for dealing with a second-year course.

keywords: curriculum design, pedagogical issues, teaching strategies

> *I find the emotional tone of this attack as disquieting as the "scientific" analysis. How many poor, innocent, novice programmers will feel guilty when their sinful use of* ***go to*** *is flailed in this letter*?—John R Rice, in response to Dijkstra's "Go To Considered Harmful" letter, *CACM*, vol. 11 no. 8. August 1968, p. 538 (reprinted vol. 26 no. 1 January 1983, p. 74*)*

1. Introduction

Conventional wisdom in Computer Science curriculum design is that the "basics" start with programming, especially coding skills – though there is a growing debate about whether programming is really the "first" skill in Computer Science [Gersting et al. 2001]. Concepts like abstract data types, object-oriented programming, and so on are "advanced" skills that come later. Even texts which provide an "objects-first" approach [Decker and Hirshfield 1995, Niño and Hosch 2002] tend to assume that the easier skill is designing your own classes, rather than using class libraries. Perhaps part of the problem is that object-oriented approaches have been slow to enter the mainstream: certainly

---

* This paper is based on experiences when the author was at the School of Computer Science at University of the Witwatersrand in South Africa

some earlier object-oriented books were centred on learning to use class libraries [Goldberg and Robson 1983], long before object-oriented programming became commonplace. Perhaps the problem is that such books are considered "advanced", or perhaps the problem is that insufficient experience has been built up of introducing object-oriented concepts for beginners. Whatever the cause of the problem, it is worth investigating introduction of programming concepts in a more object-oriented way, to provide an additional data point for curriculum designers.

Further, the Curriculum 2001 [ACM and IEEE Computer Society 2001] standard has endorsed an approach similar to that described here as a valid introductory approach, so it is useful to make the experiences described here available to those considering implementing an objects-first strategy.

Should a strongly objects-first approach be harder for beginners?

This paper argues the contrary: that Bloom's Taxonomy [Bloom 1956] places relatively factual knowledge low on the skills hierarchy—knowledge such as knowing interfaces of a standard class library. Skills higher up the hierarchy are needed for building programs from scratch. Hence, an objects-first approach can potentially be easier for beginners.

The approach advocated here is one of focusing early on the factual content of the course without spending a lot of time on programming; programming skills are developed later. This ordering may appear to be "backwards"—hence the title of the paper.

In previous work, it has been argued that tools and concepts developed to support abstraction are meant to make programming easier—yet we do them late, and the result is that students become stuck on "bad" ways of programming taught early in the curriculum [Machanick 1998a]. The result? Practitioners widely report that techniques like reuse of classes and going to a library, before rolling your own class, are hard to achieve in practice [Auer 1995; Berg *et al*. 1995; Fayad and Tsai 1995; Frakes and Fox 1995]. Should this be surprising? After all, we no longer teach the **goto** as a lower-level construct before we teach loops. Dijkstra's letter to Communications of the ACM arguing that **goto** was harmful was not immediately accepted, though it is now almost universal wisdom over thirty years after March 1968. Certainly, before **goto** was removed from the standard programming lexicon, it was widely believed among educators that constructs like loops were "easier" to understand if built up out of the "more primitive" **goto**. Not many people would argue that case today (though there is a school which argues for a lower-level constructs first approach [Patt and Patel 2001]), yet a similar debate is now beginning around object-oriented programming. It is argued that algorithms and data structures are more fundamental than classes (since classes are built up out of algorithms and conventional data structures), so classes cannot be taught unless a conventional algorithms and data background is done first.

Much argument around curriculum issues is subjective. A clear problem is that those who have the most interest in investigating a given approach are likely to be advocates. If an advocate teaches using their preferred style, and then measures the result, it's hard to claim objectivity. Be that as it may,

measurement must start somewhere. Once advocates have made their case strongly enough, others will try the approach, and less subjective evaluation can begin.

What is the case so far for a strongly objects-first approach?

Earlier work on reordering has been done at second-year level. A course called "Advanced Programming" (AP) was redesigned as an object-oriented course called "Data Abstraction and Algorithms (DAA). AP was originally taught using Pascal, and later migrated to Modula-2. While the Modula-2 version introduced some object-oriented concepts, general coverage of abstraction was weak because standard Modula-2 does not include classes, and its abstraction mechanisms are poor. The DAA course, while successful in the sense of covering a lot more ground than was previously covered, was somewhat hobbled by difficulties of the language used, namely C++.

Since the language was a problem in DAA, it was hard to make strong claims about the utility of the approach, other than to observe that shifting from a less strongly objects-first approach (also in C++) made it possible to cover many more concepts without losing the students [Machanick 1998*a*].

In principle, however, it seemed that the ideas which had worked in DAA could be used in a more introductory course, provided that a better language could be used. In particular, a language without the semantic baggage of C++, but with a stronger standard class library would in principle work well.

This paper describes a newer course, Data and Data Structures (DDS), which was put together as an introductory course for a Higher Diploma class. Although the students all already had a degree, most had not had significant prior exposure to computers, so this course was essentially introductory. The approach taken was similar to that of the DAA course, except that the libraries available in Java made it possible to place more emphasis on standard application programming interfaces (APIs), a central facet of encouraging reuse. The Java 2 Platform (also called JDK 1.2) offers a reasonable container class library, which made for a much more convincing approach to introducing the students to reuse through abstract data types.

As compared with previously published work [Machanick 1998*a*], the DDS course explores how to use Bloom's Taxonomy in curriculum design, specifically in a particular course—as opposed to an earlier proposal to use Bloom's Taxonomy in the overall design of a course [Machanick 1998*b*]. Bloom's taxonomy has been most commonly used in pre-university curriculum design, but there is a growing movement to include it in higher education curriculum design; the idea

The remainder of this paper describes experience with the DDS course, in the following order. Section 2 describes the background to the Higher Diploma DDS course. Section 3 presents the design philosophy of the course. Section 4 provides some practical experiences. Section 5 evaluates the course, and Section 6 provides some conclusions.

2. Background to the Higher Diploma DDS Course

At the School of Computer Science, at the University of the Witwatersrand (usually shortened to Wits) in South Africa, Computer Science forms part of a three-year BSc degree. Students have the option of taking an additional one-year degree, BSc Honours, an intensive junior graduate-level programme.

The Higher Diploma in Computer Science (HDipCS) is offered as a bridging programme for students who have no background in Computer Science, or significantly less than is offered in the Wits degree, but who already have a degree. The other significant entry requirement is some background in mathematics (at the equivalent level to the Wits first year of mathematics). Students typically do the HDipCS for one of three reasons:

- their first degree was not sufficiently "marketable"
- they want a combination of skills from more than one discipline
- they are interested in doing a higher degree in Computer Science but lack the background to be accepted into a quality programme

The HDipCS is very popular: the School had around 120 applicants for 30 places in 1999. However, the HDipCS presents significant logistical problems for the School. The School does not have the resources to run it as a separate program, and the course is put together out of most of the undergraduate curriculum, done in one year. Since the curriculum covers a large fraction of the undergraduate courses, taken over one year, a logical sequence of courses is hard to piece together, given rules for prerequisites. Consequently, in the past, it has been necessary to offer repackaged versions of a few of the regular courses specifically for the HDipCS class.

For 1999, a new approach was tried, in which several courses were changed in terms of the ordering. The HDipCS class did all their lectures with the regular undergraduate class, except for one course to bring them up to speed, so that they should be able to take on the remainder of the curriculum in a less-than-ideal order.

An appendix illustrates the timing of the 1998 and 1999 courses, showing how much more "efficient" the 1999 course is in terms of requiring new resources, though at the expense of doing components in a less optimal order.

A particular problem was that the second-year Data Abstraction and Algorithms (DAA) course, which covered object-oriented concepts in C++, along with algorithm analysis, was a relatively heavy-weight course, yet it could not be too late in the year, because it was a prerequisite for other second-year courses. In 1998, the problem had been solved by running a special version of DAA for the HDipCS class; in 1999 the plan was to do away with the need for such separate versions of courses wherever possible.

The solution was to move a first-year course, Data and Data Structures (DDS), to earlier in the year, and to change its content (borrowing the name from an existing course allowed us to avoid a rule change). It became the one course that was put on specifically for the HDipCS class—and, not only

that, had different content to the course given to BSc students. The course was started 3 weeks before the normal start of term, to allow time for concepts to be absorbed, before the class started more advanced courses.

3. Design Philosophy of the Higher Diploma DDS Course

3.1 Introduction

The approach used in the course was novel in several respects.

First, in terms of the order of cognitive skills, the approach was based on starting with factual skills, and moving on to application, an approach advocated in earlier work for overall design of the Computer Science curriculum [Machanick 1998*b*]. Second, the abstraction-first approach used in the DAA course [Machanick 1998*a*] was followed.

Let's consider a bit more detail of these two ideas—then conclude the section by considering whether the approach is really "backwards".

3.2 Low-Level-Cognition First

In education theory as applied in schools, Bloom's Taxonomy [Bloom 1956] is widely used; in recent times, Bloom's Taxonomy has also started to receive attention in higher education (for example, in the ACM/IEEE Curriculum 2001 process, in which the author is involved).

Bloom's Taxonomy divides skills into those at a beginner's cognitive level, through to higher-level abilities. One of the strengths of this classification of skills is that a taxonomy, unlike a straightforward classification, is rooted in an objectively-determined framework, whereas a classification in its more general sense may be based on arbitrary criteria. Bloom's Taxonomy is founded in extensive research and surveys of educators. Accordingly, it is a useful framework for judging the appropriateness of a given kind of task for a given skill level.

Much of the material in Bloom's report relates to artistic or creative work but relatively minor adaptation makes the ideas suitable for Computer Science

The taxonomy orders skills as follows, from lowest cognitive skills to highest:

- knowledge—factual knowledge
- comprehension
- application
- analysis
- synthesis
- evaluation

While other breakdowns are possible (and the authors of the original report acknowledge this), it is useful to use this broad breakdown as a basis for examining any curriculum in terms of the order it presents material and the demands it makes of students at each stage of their studies. Any deviation

from the order suggested by Bloom's Taxonomy can of course be justified, but it is a useful start to compare skills against such an accepted taxonomy, to reveal any major problems in the way a curriculum is structured.

It is argued elsewhere [Machanick 1998*b*] that Bloom's Taxonomy should be used as a basis for a more complete overhaul of which topics are taught where across the entire curriculum. Here, Bloom's Taxonomy is used as a basis for ordering concepts within one course.

This approach of using Bloom's Taxonomy to design curriculum is a slight variation on the traditional use of the taxonomy, which is its use to design questions to test particular cognitive skills. Bloom's report suggests styles of questions to test knowledge at each level. However, it is an unnecessarily narrow reading of the report to restrict its purpose to providing guidance as to how to set questions of a given level of difficulty.

The approach used is to start with the relatively factual aspects of the course first. Ideas like layers of virtual machines are presented in lectures, and demonstrated in the lab. Areas where higher-level cognition is exercised, including design skills, are put as late as possible.

Compared with a conventional view of teaching programming, many concepts are taught in the opposite order to the standard order. Interfaces to class libraries come before detail of how to code a class, because the first is "knowing" while the latter requires "application" in simpler cases, and "analysis" or even "synthesis" in more complex cases.

The knowledge-before-application approach was applied by doing all the lectures in the first three weeks, while moving to using laboratory and tutorial sessions in the last 4 weeks. The goal was to present a factual view of the subject first, followed by increasing requirements on the students to apply their knowledge, and later, to use higher-level cognitive skills required for making design decisions.

The lectures were presented as 3 double-lecture sessions over three days for the first three weeks, and the class did two laboratory sessions, with a tutorial between the sessions. For laboratory work, the class was split in two, to facilitate communication with tutors. At this stage of the course class was doing no other courses, and were expected to devote their undivided attention to the course. For this initial period, 3 tutors were used: two who had just finished their BSc, and one MSc student. To some extent, the approach was adopted on the basis that lectures are a relatively inefficient way of imparting information—so why not blow them away fast, and focus attention instead on the more effective strategies?

During the last 4 weeks, the focus shifted from relatively small additions to programs to timing programs to see if timing matched predictions of algorithm analysis. Finally, the course concluded with a relatively simple programming exercise, in which the solution totaled about 120 lines of code, in which the class had to develop all the code (excluding libraries). In this final phase, two additional tutors were added: an Honours student, and an additional MSc student, who has an interest in both

Java and curriculum issues. This strong tutor support made for a quality learning experience in the final 4 weeks. The final four weeks overlapped other courses, and the pattern of laboratory-tutorial-laboratory was maintained.

While the final programming exercise was relatively simple, it relied on many concepts not commonly found in introductory courses: using a standard API, designing a program so that the major data structure could easily be changed, using classes and objects, and working to a design.

Earlier stages of the course, where more emphasis was placed on *reading* code involved significantly more sophisticated examples, including an applet which uses a naïve convex hull algorithm to draw a polygon around a set of points created by clicking with the mouse.

3.3 Abstraction First

The abstraction-first approach works by hiding detail until students are ready for it. It starts with introducing interfaces to class libraries as the first stage of understanding programming, and works towards understanding how to implement programs, finally ending with implementation from scratch (e.g., if you need to implement a new library).

Since the class was mostly not previously exposed to computers, the abstraction-first approach was extended to covering higher-level virtual machines down to programming. This introduction was presented relatively factually, without much on how the various levels of abstract virtual machine work. Instead, until programming concepts were introduced, the layers were demonstrated at the user level.

The order in which concepts were introduced was as follows:

- applications and user interfaces (including consistent virtual machines, using the Macintosh interface as an example)
- operating systems and networks, including servers
- computer architecture, at a very conceptual level (an introductory computer organization course followed soon after this course)
- programming tools including role of compilers linkers, integrated development environments (IDEs), application programming interfaces (APIs: a fancy name for class libraries) and runtime environments
- classes and object-oriented concepts
- use of APIs
- role of efficiency in making design decisions (algorithms and data structures)
- role of data structures

- the Java 2 Collections API
- more detail on complexity

Unlike with the older second-year DAA course, where C++ made the ordering awkward, Java worked well for this ordering.

3.4 Is this Backwards?

In the sense that developing a relatively simple program from scratch happened last, after exploring concepts like data abstraction and algorithm analysis, the order of concepts in the course may seem backwards relative to a "conventional" introductory course. However, strongly object-oriented languages with strong support for abstraction, especially languages like Java and Smalltalk, make a relatively abstract starting point less unnatural than for example with a language like C++ which exposes its guts to the programmer.

C++ made the ordering in the older second-year DAA course awkward because it was hard to avoid low-level issues early, like the semantics of copying pointers. Java and similar object-oriented languages, with their hidden pointers and garbage collection, strong and relatively easy-to-understand class libraries and simple syntax, make the abstraction-first approach more natural.

Compare the proposal here with the development of structured programming, where the FORTRAN advocates fought a rearguard action for the **goto**, monolithic main programs with minimal procedural decomposition and simple data types. Even those who admitted that there were better program structuring techniques argued that the skills required were "advanced" and should be taught later. How many people would argue today that you need to learn the **goto** before you can understand loops?

4. Practical Experiences

4.1 Introduction

There were a number of novel aspects to the course, which makes it difficult to evaluate as a whole without confusing results of several variables. This section gives a subjective report on experiences with the course, including the method of assessment, the use of Java and timing of the course.

4.2 Assessment

Assessment was carried out in two forms: small tests to measure progress, and larger tests to evaluate comprehension in the lab. The School has for some time advocated testing practical skills in tests rather than in assignments. While there are assignments, they are not directly assessed but rather in a test after the end of the assignment [Hazelhurst and Mueller 1989]. The value of this approach is it separates out the learning aspect of the lab, where making mistakes is good, from assessment, where

"getting it right" is the goal. Also, it becomes easier to encourage group work, without the complications of assessing individual performance of group work.

Regular small tests, ranging from 1% to 3% of the total credit for the course, were used to measure students' progress. Students were also provided with a self-assessment sheet for each laboratory session, so that they could evaluate for themselves whether they had learnt what was required from the session. With two laboratory sessions and a tutorial per week, as well as 2 short multiple-choice tests every week (barring weeks with bigger tests), the students had rapid feedback on their problems. Furthermore, the rapid feedback made it possible to adjust expectations of the class for each laboratory session and test.

An effect of this kind of micro-assessment was that students' results steadily improved as we tested a section of the work, as they had the opportunity to work over related material soon after seeing how well they had understood earlier concepts. While successive tests did not overlap, the time lapse between two tests was short enough that the kind of concept tested was not very different from the previous test.

4.3 Java Experiences

The use of Java also worked well, though using the latest libraries did cause problems. The available equipment was 5-year-old Macintoshes. While Macintoshes have many benefits, being in the forefront of Java availability is not one of them. It was necessary to hack library files to run the latest Java APIs, including the Collections container class library. However, it was worth the effort, as Collections is cleanly designed, and provides a much stronger vehicle for illustrating principles of data structures using abstract data types than the C++ Standard Template Library (STL).

What's more, it was possible to introduce concepts in a natural order (given the abstraction-first strategy), without nearly as many diversions and cross-references, as was required in C++ [Machanick 1998*a*]. For example, in C++, anything needing dynamic data structures requires introduction of pointers. It is possible to hide pointers to some extent, but it is difficult to avoid complications arising from the complex semantics of copying pointers.

A final bonus from using Java was that tutors were highly motivated, given that it was seen to be a leading-edge language. Usually, introducing a new language causes problems with tutors, as they have no prior exposure to the language and teaching issues that are language-specific. The opposite effect was experienced in this course. Part of the reason for this effect was that the tutors had all previously been exposed to the underlying concepts in the second-year DAA course (some had taken the course two years previously; another had previously tutored the course). They were not only able to appreciate the intent of the approach, but to see that it worked better with a language with cleaner semantics.

Particular features in Java's favour included:

- no explicit pointers (including garbage collection for dynamic data)

- strong object-orientation—although primitive types are available as well, it is possible to program in Java only using classes and objects
- good standard libraries—although some of the Java APIs are a bit complex for an introduction, it is possible to do a reasonable course using a selection of APIs, especially the simpler graphics-oriented class libraries and the Collections classes (which provide a simple introduction to abstract data types)
- easy to do fun examples—related to the good standard libraries: it is not hard to demonstrate a simple graphical application or applet in a web page

Given more up to date equipment (the Macintoshes were 1995 vintage), it would also have been possible to exploit Java more fully, e.g., using web-based course tools.

4.4 Timing

The approach of running all the lectures in three weeks required concentrated effort, and was at first unsettling for the students, as they did not have the time to grasp the concepts as they were presented. However, the repeated reinforcement through laboratory sessions, tutorials and tests worked well.

The underlying idea was to present the course relatively factually first, then build on the factual base by requiring increasingly high-level cognitive skills as the course developed. This approach appeared to work in that students were adequately prepared for the second-year DAA course which they took next.

5 Evaluation of the Higher Diploma DDS Course

Since a major challenge in designing the course was to prepare the class for the later C++-based DAA course, it is useful to consider how well they performed in DAA. Since the DDS course was most focused on introduction of abstraction and object-oriented concepts, it should be expected that the class should do better in the parts of DAA which covered the areas most emphasized in DDS.

year		*class mark %*	*exam %*	*overall %*	*% failed*	*class size*
1998	CS II	57.7	56.2	56.9	31.3	96
	HDipCS	54.3	56.5	55.5	42.9	14
1999	CS II	57.8	54.3	56.0	29.7	74
	HDipCS	59.1	53.7	56.4	25.0	20

Table 1. HDipCS Success Rate versus Computer Science II in DAA

marks are averages for each class; 50% is a pass; final mark is (class mark + exam)/2

year	number at end	passed	failed	% passed
1995	11	6	5	55
1996	27	20	7	75
1997	16	13	3	81
1998	17	13	4	76
1999	21	18	3	86

Table 2. Statistics on overall HDipCS Success

By the end of the DDS course, 21 students had written the examination, of whom 16 had passed (76%). Of further interest is the fact that a relatively high fraction of the class passed at the end of the year, despite the non-optimal ordering of topics (particularly as compared with previous years).

Table 1 presents figures on the relative success rate of the class versus the second-year Computer Science class in the same year. Both groups of students attended the same lectures, so there is no variation in what or how they were taught. Table 2 presents the overall figures for each year from 1995 to 1999, showing that the 1999 class had a good final pass rate.

Since class sizes are not very big and students are drawn from very diverse backgrounds, not too much can be read into these numbers. It is possible, for example, that some of the variation across years occurred because the 1999 class was unusually good, a factor which can only be taken into account by experimenting across multiple years. Unfortunately, a directly comparable experiment is not possible because the Wits Computer Science second-year curriculum changed significantly in 2000.

However, there are a number of factors which should have made the DAA course more difficult for the class:

- the ordering of topics was worse than in previous year: given the compromises made to reduce the number of courses which would have to be put on specially for the HDipCS group, their background by the time they started doing the DAA course was not as strong as it should have been
- the class did not have a strong introduction to algorithms before DAA; consequently, the only test on which they did significantly worse than the second year class was a big test, about halfway through the DAA course, on algorithms

- the year before, the HDipCS class was smaller, and was lectured to as a separate group when doing DAA; in 1999, they took DAA as part of the larger Computer Science II (second year) class, with no special allowance made for their different background

- in 1999, the DAA course was started at a time when the HDipCS group had much less programming background than the HDipCS class of 1998: the 1998 class had an introduction to Pascal programming before official courses started, as well as two first year courses, fundamental algorithmic concepts (FAC) and data and data structures (DDS) before starting DAA

Despite these factors, as can be seen in Table 1, the 1999 HDipCS class did better in relation to the Computer Science II class, than the HDipCS class of 1998. It is therefore reasonable to conclude (if taking into account the problems in setting up an objective experiment which have already been noted) that the new DDS course was a reasonable background for an advanced data abstraction and algorithms topic.

A lecturer evaluation using a standard university survey was above average, a very good result for a new, experimental course. Particularly encouraging was the fact that an unusually high fraction of the students in the class was positive on wanting to study the subject further. Most of the points which scored significantly below average related to lecturing, not surprising given that the lectures were fast-paced. However, it should have been possible, had the course been run again on the same basis, to improve on the rating for lecturing.

6. Conclusions

The course at first appraisal appears to have gone well. The class was strongly motivated, given that they are highly selected, and worked hard. While the order of concepts was unusual, the only real problem presented by the unusual ordering was that the class had to rely on the supplied notes (totaling 240 pages) and could not find other material that went in a similar order. Results were better than in previous years, though it would have been useful to repeat the experiment on a different group of students.

It is reasonable to conclude based on the evidence presented here that the “backwards” approach is a viable alternative, particularly when supported by a suitable language. In particular, that an objects-first or abstraction-first approach is suited to quick presentation of factual content, before switching to higher cognitive skills.

It would however be more difficult to arrive at a stronger conclusion, such as that the “backwards” approach is more effective than a conventional approach to introducing programming, given the experimental limitations of this study. However, the data presented here makes a case for further research.

Despite the relative success of the course, some improvements could easily be made.

If run again, the lectures could be made more appealing, with more time allowed for presenting examples, and describing the focus of each specific lecture. It would also be useful to have a stronger version of Java, with more up to date equipment. Unfortunately, subsequent changes in other aspects of the curriculum have made it difficult to repeat the experiment under like conditions.

Despite teething problems inevitable with a new course, equipment problems and the difficulties of dealing with a class of extremely wide backgrounds, this course was popular with students and tutors.

Positive experience of the new DDS course suggests that the second-year DAA course could be much improved with some of the ideas presented here. The positive experience also suggests trying the same approach more broadly.

The strategy of an intensive lecture-based phase followed by more leisurely laboratory and tutorial work would be difficult to apply to other classes, though some attempt is being made to adapt the approach to other courses [Machanick 2000]. The HDipCS programme, since it is purely run in one School, made it possible to run an unusual schedule. The undergraduate programme , by contrast, has to fit with other courses, so such a change would not be as easy to accommodate. Other institutions would likely run into a similar problem, unless all courses students took were restructured in this way (i.e., the students' overall timetable would be designed around a few weeks of intensive lectures, followed by laboratory and tutorial sessions). The approach advocated here could work particularly well for part-time courses given through distance education: intensive lectures could be presented by television or via the Internet, and the follow-up phase of laboratory work could be done by students taking some time off work to attend a laboratory resource centre.

The lecture notes form the basis for a planned book. It may be difficult to find a publisher given the novelty of the approach, but classroom experience suggests that teaching backwards works. However, the acceptance of the underlying strategy as part of the Curriculum 2001 standard may help to mainstream the approach.

Acknowledgments

The new DDS course would not have run as smoothly without the enthusiastic support of tutors, especially Kim Mason, Lucy Kaschula, Shane Morton and Adi Attar. Ian Sanders provided helpful comments on an early draft of this paper.

References

[Auer 1995]. Ken Auer. Smalltalk Training: As Innovative as the Environment, *Comm. ACM*, vol. 38 no. 10 October 1995, pp. 115–117.

[Berg *et al*. 1995]. William Berg, Marshall Cline and Mike Girou. Lessons Learned from the OS/400 OO Project., *Comm. ACM*, vol. 38 no. 10 October 1995, pp. 54–64.

[Bloom 1956] Benjamin S Bloom (ed.). *Taxonomy of Educational Objectives: Book 1 Cognitive Domain*, Longman, London, 1956.

[ACM and IEEE Computer Society 2001] *Computing Curricula 2001*, ACM and IEEE Computer Society, http://www.computer.org/education/cc2001/report/, March 2000.

[Decker and Hirshfield 1995] Rick Decker and Stuart Hirshfield. *The Object Concept*, PWS, Boston, MA, 1995.

[Fayad and Tsai 1995]. Mohamed E Fayad and Wei-Tek Tsai. Object-Oriented Experiences, *Comm. ACM*, vol. 38 no. 10 October 1995, pp. 51–53.

[Frakes and Fox 1995]. Frakes and Fox. Sixteen Questions About Software Reuse, *Comm. ACM*, vol. 38 no. 6 June 1995, pp. 75–87,112.

[Gersting *et al*. 2001] J Gersting, PB Henderson, P Machanick, YN and Patt. Programming early considered harmful, *Proc. SIGCSE-2001*, Charlotte, NC. 2001, pp 402-403.

[Goldberg and Robson 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Hazelhurst and Mueller 1989] Scott Hazelhurst and Conrad Mueller. Assessment of Programming Assignments, Technical Report TR-Wits-CS-1989-05, School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, 1989.

[Machanick 1998*a*] P Machanick. The Abstraction-First Approach to Data Abstraction and Algorithms, *Computers & Education*, 1998, vol. 31 no. 2, September 1998, pp. 135-150.

[Machanick 1998*b*] P Machanick. The Skills Hierarchy and Curriculum, *Proc. SAICSIT '98*, Gordon's Bay, November 1998, pp. 54-62
(also at `http://www.itee.uq.edu.au/~philip/Publications/SAICSIT/skills-98.html`).

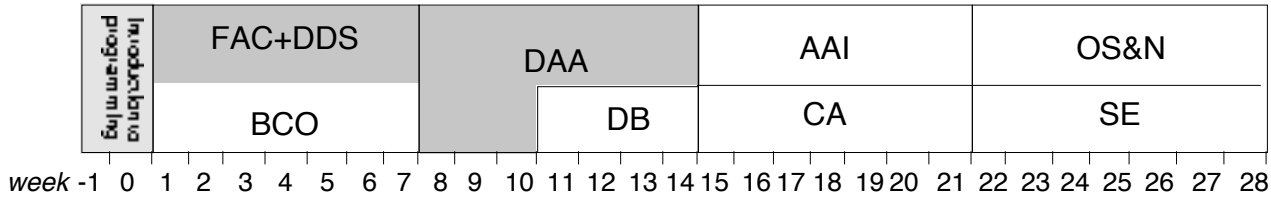
[Machanick 2000] P Machanick. Experience of Applying Bloom's Taxonomy in Three Courses, *Proc. Southern African Computer Lecturers' Association Conference*, Strand, June 2000, pp 135-144
(also at `http://www.itee.uq.edu.au/~philip/Publications/SACLA/blooms2k-abstr.html`).

[Niño and Hosch 2002] Jaime Niño and Frederick A Hosch. *Introduction to Programming and Object-Oriented Design using Java*, Wiley, New York, 2002.

[Patt and Patel 2001] Yale N Patt and sanjay J Patel. *Introduction to Computing Systems: from Bits & Gates to C and Beyond*, McGraw Hill, Boston, 2001
(also at `http://www.itee.uq.edu.au/~philip/Publications/SACLA/blooms2k-abstr.html`).

Appendix: Higher Diploma: 1998 versus 1999

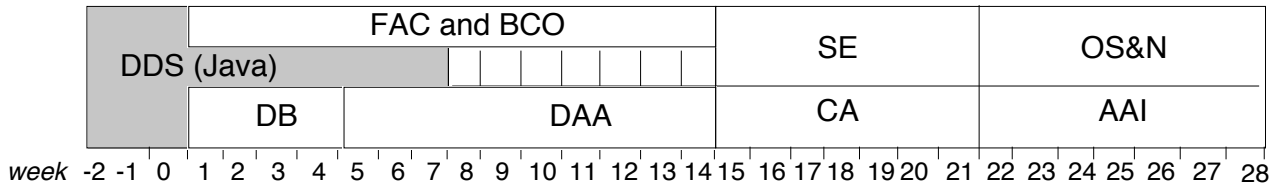
The Higher Diploma was arranged as follows in 1998:



Shaded blocks are courses put on specifically for the higher diploma; those starting from week 1 were based on courses given to the undergraduate classes, but where the standard order was not suitable (so special versions had to be run for the HDipCS). Before the start of the course, an introduction to programming in Pascal was given over two weeks, but not for credit. Course names are as follows (with year of study in the conventional 3-year BSc degree):

- FAC (1st year): fundamental algorithmic concepts: introduction to algorithms and algorithm analysis
- DDS (1st year): data and data structures: introduction to data structures including more algorithm analysis (for the HDipCS, a sped-up version of FAC and DDS was given as one course)
- DAA (2nd year): C++-based data abstraction and algorithms
- DB (2nd year): database
- AAI (3rd year): algorithms and artificial intelligence
- CA (2nd year): computer architecture
- OS&N (2nd year): operating systems and networks
- SE (3rd year): software engineering

Observe the changes in the 1999 version:



Notice specifically how much less of the course is specific to the HDipCS class. Also note that the DAA course starts after far less time for establishing background. (The BCO and FAC courses are shown as a lower longer box because in 1999, they were taught at half speed over twice the number of weeks.)