

Teaching Programming Backwards

Philip Machanick
Department of Computer Science, University of the Witwatersrand
Private Bag 3, 2050 Wits, South Africa
philip@cs.wits.ac.za

Abstract

Conventional wisdom places programming early in the curriculum, and places emphasis on basic coding skills before “advanced” subjects like abstract data types, object-oriented programming, etc. This paper presents an experience of doing things backwards: students are exposed first to abstraction, then to classes, objects and application programming interfaces (APIs) and finally to coding. Java turns out, in its latest form with much-improved libraries, to be well-suited to this approach.

1. INTRODUCTION

Conventional wisdom in Computer Science curriculum design is that the “basics” start with programming, especially coding skills, and concepts like abstract data types, object-oriented programming, etc. are “advanced” skills that come later. Previously, 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 etc. is hard to achieve in practice [Auer 1995; Berg et al. 1995; Fayad and Tsai 1995; Frakes and Fox 1995].

Earlier work on reordering has been done at second-year level, where a course previously called “Advanced Programming” (AP) was redesigned as an object-oriented course called “Data Abstraction and Algorithms (DAA). This 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, C++.

This paper describes a new 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 APIs, a central facet of encouraging reuse. The Java 2 Platform (also called JDK 1.2) offers a good container class library, which made for a much more convincing approach to introducing the students to reuse through abstract data types.

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 Department of Computer Science, University of the Witwatersrand, a Higher Diploma in Computer Science (HDipCS) is offered as a bridging course for students who have no background in Computer Science, or significantly less than is offered in our degree, but who already have a degree. The other significant entry requirement is a year’s mathematics (at the equivalent level to our first year of mathematics). The HDipCS is very popular: we had around 120 applicants for 30 places in 1999. However, the HDipCS presents significant logistical problems for the department. We do 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 is hard to piece together, given prerequisites. Consequently, in the past, it has been necessary to offer versions of the regular courses at different times of the year for the HDipCS class.

For 1999, a new approach was tried, in which several courses were changed in terms of the ordering, so that the HDipCS class does 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.

A particular problem is that the second-year DAA course which covers object-oriented concepts in C++, along with algorithm analysis, is a relatively heavy-weight course, yet it cannot be too late in the year, because it is a prerequisite for other second-year courses.

The solution was to move the first-year Data and Data Structures (DDS) course to earlier in the year, and to change its content, so that it is the one course that is put on specifically for the HDipCS class—and not only that, has different

content to the course given to BSc students. The course was started 3 weeks before 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

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 on a wider scale for curriculum design [Machanick 1998*b*]. Second, the abstraction-first approach used in the DAA course [Machanick 1998*a*] was followed. 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. The order was:

- applications and user interfaces (including consistent virtual machines as in the Macintosh interface)
- operating systems and networks
- computer architecture
- programming tools including role of compilers linkers, integrated development environments (IDEs), APIs 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

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 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.

In the sense that developing a relatively simple program from scratch happened last, after exploring concepts like data abstraction, the order of concepts in the course may seem backwards relative to a “conventional” introductory course. However, when using object-oriented languages, especially languages like Java and Smalltalk which come with good libraries and which hide less understandable aspects of semantics (especially pointers and dynamic memory management), a relatively abstract starting point is less unnatural than for example with a language like C++ which exposes its guts to the programmer.

| 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 |

Table 1. Statistics on HDipCS Success

4. PRACTICAL EXPERIENCES

The real test of the course is how well the class does in the overall HDipCS programme. However, at time of writing, when results totaling 28% of the course were available, 24 of the original 32 students accepted for the course were still registered, and of these, 22 were passing, with none with a result below 47%.

What was particularly useful in assessing progress was running regular small tests, ranging from 1% to 3% of the total credit for the course, as well as providing the students 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.

The use of Java also worked well, though using the latest libraries did cause problems. The available equipment was 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++ [Machanic 1998a].

A final bonus from using Java was that tutors were highly motivated, given that it was seen to be a leading-edge language.

Given more up to date equipment (the Macintoshes are 1995 vintage), it would also have been possible to exploit Java more fully, e.g., using web-based course tools. Others have used WebCT, but with mixed results, as the machines are really too slow and have too little memory to run a web browser and a development environment simultaneously.

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.

5 EVALUATION OF THE HIGHER DIPLOMA DDS COURSE

As indicated before the results of the course cannot be evaluated on their own: the big test really is how well the class copes with the C++-based DAA course. If they can cope with that, despite the fact that they have not had a full first-year-level coverage of data structures and algorithms, the DDS course can be judged to be a success.

The pass rate contrasts with previous runs of the course. At time of writing, 22 out of 24 (92%) were passing, out of the 30 who started the course. Only one of the 6 who dropped out was doing badly, and even this one was doing badly for reasons of personal problems. The rest of those who dropped out did so for reasons unrelated to performance, such as being accepted to medical school, or deciding that Computer Science was not what they thought it was. The drop-out rate is typical of previous runs of this course, and the pass rate is higher than in previous years, though the real test will come when the class does other courses.

By way of comparison, in previous years, numbers were as in Table 1 (though it should be noted that final pass rates can only really be compared at the end of the year). Note that the "number at end" figure does not include students who were accepted and dropped out or did not show up, and is therefore comparable to the figure 24 in the class of 1999. Unfortunately we do not have accurate figures on the number who started in previous years.

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 number 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.

Further evaluation will be possible at the end of the year, when the class finishes.

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 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 have been better than in previous years, though proper evaluation will require consideration of their results in other courses.

If run again, the lectures could be made more appealing, with more a 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.

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.

Although not absolute evidence of the success of the approach, subjective impressions to date suggest that the second-year DAA course could be much improved with some of the ideas presented here.

While the approach has some general applicability, the strategy of an intensive lecture-based phase followed by more leisurely laboratory and tutorial work would be difficult to apply to other classes. The HDipCS programme, since it is purely run in one department, made it possible to run an unusual schedule.

The lecture notes form the basis for a planned book: see the appended table of contents. It may be difficult to find a publisher given the novelty of the approach, but classroom experience suggests that teaching backwards works. This approach to teaching lends new meaning to the expression “to know a subject backwards”.

Acknowledgments

This course would not have been possible without the enthusiastic support of tutors, especially Kim Mason, Lucy Kaschula, Shane Morton and Adi Attar.

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.
- [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.
- [Machanick 1998a] P Machanick. The Abstraction-First Approach to Data Abstraction and Algorithms, *Computers & Education*, 1998, vol. 31 no. 2, September 1998, pp. 135-150.
- [Machanick 1998b] P Machanick. The Skills Hierarchy and Curriculum, *Proc. SAICSIT '98*, Gordon's Bay, November 1998, pp. 54-62.

APPENDIX: DDS CONTENTS

Chapter 1 Introduction

- 1.1 Introduction
- 1.2 Major Concepts
- 1.3 Specific Examples
- 1.4 Road Map
- 1.5 Further Reading
- 1.6 Exercises

Part 1 Overview of Computer Systems

Chapter 2 Virtual Machines

- 2.1 Introduction
- 2.2 Computer Architecture
- 2.3 Operating Systems
- 2.4 Other Shared Resources
- 2.5 Programmer's Virtual Machine Layers
- 2.6 Applications
- 2.7 Further Reading
- 2.8 Exercises

Chapter 3 Where Programming Fits In

- 3.1 Introduction
- 3.2 Algorithms and Data Revisited
- 3.3 Efficiency
- 3.4 Classes, Objects and APIs
- 3.5 Applications and Applets
- 3.6 Putting it all Together: Java
- 3.7 Further Reading
- 3.8 Exercises

Chapter 4 Programming Tools

- 4.1 Introduction
- 4.2 Development Environments
- 4.3 Editors
- 4.4 Compilers
- 4.5 Linkers
- 4.6 Debuggers
- 4.7 Runtime Environments
- 4.8 Putting it all Together: Java
- 4.9 Further Reading
- 4.10 Exercises

Part 2 Object-Oriented Programming

Chapter 5 Anatomy of a Java Program

- 5.1 Introduction
- 5.2 Where a Program Starts
- 5.3 Files Making Up a Java Program
- 5.4 What Goes Into a Java Program
- 5.5 Examples Revisited
- 5.6 Relationship to Programming Tools
- 5.7 Further Reading
- 5.8 Exercises

Chapter 6 Objects and Classes

- 6.1 Introduction
- 6.2 What a Class Defines
- 6.3 Life Cycle of an Object Revisited
- 6.4 Instance vs. Class Properties
- 6.5 Collections Revisited
- 6.6 Some Loose Ends
- 6.7 Further Reading
- 6.8 Exercises

Chapter 7 Common Java APIs

- 7.1 Introduction
- 7.2 Abstract Windowing Toolkit (AWT)
- 7.3 Applet
- 7.4 The `java.io` Package
- 7.5 Collections
- 7.6 Putting it all Together: Fence Again
- 7.7 Further Reading
- 7.8 Exercises

Part 3 Use of Data And Data Structures

Chapter 8 Data and Data Structures

- 8.1 Introduction
- 8.2 General Overview of Data Structures
- 8.3 General Issues for Containers
- 8.4 Kinds of Container
- 8.5 Relation to Java Collections
- 8.6 Putting it all Together: No Duplicates
- 8.7 Further Reading
- 8.8 Exercises

Chapter 9 Java Collections

- 9.1 Introduction
- 9.2 General Overview of Data Structures
- 9.3 Design Principles of Collections

- 9.4 General Overview of Collections
- 9.5 Collections Algorithms
- 9.6 Design Choices
- 9.7 Efficiency Concerns: Space and Time
- 9.8 Putting it all Together: New Container
- 9.9 Further Reading
- 9.10 Exercises

Part 4 Complexity: *More Chapters Planned*

Chapter 10 Efficiency Issues

- 10.1 Introduction
- 10.2 Why Efficiency Counts
- 10.3 Computational Complexity Revisited
- 10.4 Data Structures and Algorithms
- 10.5 Where it all Fits in
- 10.6 Further Reading
- 10.7 Exercises