Title: CS1: Perspectives on Programming Languages and the Breadth-First Approach

Authors:

Richard Close, Ph.D.
U. S. Coast Guard Academy
27 Mohegan Ave.
New London, CT 06320-4195
rclose@cga.uscg.mil

Danny Kopec, Ph.D.
Brooklyn College
2900 Bedford Ave.
New York, NY 11210
718-951-5578
drk2501@aol.com

Jim Aman, Ph.D.
Columbus School for Girls
56 S. Columbia Ave.
Columbus, OH 43209
614-252-0781
jimaman@acm.org

Abstract:

With development of *Curriculum 2001* now begun, it is important not to overlook the experiences of the past. *Curriculum '91* allowed a number of approaches to the CS1 course, and they met with varying degrees of success in different environments. The authors of this paper applied a number of pedagogical ideas to this first course in computer science. Their experiences – successes and failures alike – are described and compared in this paper with the intent of triggering discussions for the *Curriculum 2001* debate.

CS1: Perspectives on Programming Languages and the Breadth-First Approach

With development of *Curriculum 2001* now begun, it is important not to overlook the experiences of the past. *Curriculum '91* allowed a number of approaches to the CS1 course, and they met with varying degrees of success in different environments. The authors of this paper applied a number of pedagogical ideas to this first course in computer science. Their experiences – successes and failures alike – are described and compared in this paper with the intent of triggering discussions for the *Curriculum 2001* debate.

Almost thirty years ago, a noted computer scientist (1) remarked that it was unfortunate that real computers had to be used in teaching computer science. Although many in the audience may have viewed this as a rather radical position at the time, it has proven to be an insightful commentary on many of our efforts to design and deliver courses in the discipline. In fact, the premise probably should be broadened to include software as well as hardware. Actual computing systems, hardware as well as software, often swamp the learner in a sea of minutia in which basic concepts are at least obscured if not completely lost. While there are difficulties in using real systems in courses at all levels, it appears that some of the greatest problems may be found at the introductory level. In particular, achieving consensus in the choice of a programming language (or none at all!) for CS1 has proven to be elusive. With *Curriculum 2001* now in the works, it is particularly timely that experience with this course be reviewed.

FROM FORTRAN TO JAVA AND BEYOND

In the early seventies, introductory courses using FORTRAN were quite common. After all, if only half in jest, it was observed that "real programmers" would use nothing else. BASIC was also popular at that time and was making inroads where time sharing was available, and the presentations were directed to more diverse audiences - perhaps not real programmers. By the end of the decade (70's), however, Pascal had captured the market. Here was a language simple, compact, and consistent which had been designed especially for education. It should be an optimum choice. Textbooks and materials using Pascal flourished. Many instructors were drawn to Pascal at all levels, but it was particularly popular in the introductory course. The Educational Testing Service (ETS) adopted Pascal for its Advanced Placement Examination. In fairness, it should be noted that there were and continue to be a number of other voices. Despite its heritage and acknowledged simplicity, Pascal can be viewed as overly complex in terms of syntax when compared with languages such as LISP (or Scheme).

It can be argued that unlike Pascal, Scheme has little if any syntax, and its semantics are based on sound mathematical principles (2). It also has become apparent that Pascal is not the *lingua franca* of professional programmers. Why should we teach something which may be irrelevant? When objects became important, efforts were made to update and enhance Pascal. Unfortunately, these efforts have resulted in more arcane systems and have been met with a rather tepid response.

Moving the language of the introductory course to C or more likely to C++ seemed imminently logical in the nineties. Almost immediately, a large number of introductory texts on C and C++ appeared, and the agendas of national meetings were filled with reports on the more or less successful implementation of C and/or C++ in CS1. It seemed that no respectable computer science department could ignore the tidal wave. ETS announced that the Advanced Placement Exam would eventually use C++. There is, however, a disturbing fact that became painfully apparent to instructors as they actually tried to present C++ and objects in their courses. Despite its highly publicized advantages, C++ is not a simple language. Also, it is now a generally accepted as an undesirable practice to simply translate code segments in an introductory text, perhaps originally written in Pascal, into C++. This lesson may not have been fully learned in that several CS1 texts have attempted to do exactly this and have appeared in several flavors (3).

The pitfalls of C for beginning students have been acknowledged for some time. The programming style of C encourages short, pithy code, replete with multiple levels of indirection. Novices also find the standard I/O facilities of C largely impenetrable. When using C++ at the introductory level, these problems remain, although the I/O difficulties are somewhat alleviated, even a cursory treatment of object principles has proven to be a formidable task. Most introductory courses using C++ are content to introduce the concept of objects and illustrate their use, while reserving object creation for a later course.

For schools determined to present a "pure" view of modularization, object-oriented programming, and good software engineering methodology in general, the emergence of ADA in the 1980's was very welcome. However, remains a favorite of only a relatively select small group of schools.

The C++ craze, however, may have been short-lived. The emergence of the Internet must be acknowledged and reflected in our courses. Recent conferences have produced an impressive number of papers on the use of Java at the introductory level. C++ is often represented as a better C. In that sense, Java may be viewed as a (slightly) simpler version of C++. While its overall utility for Internet programming may be somewhat specious, Java does project the aura of modern practice and with the implication that jobs may be available for the cognoscenti. Possibly for these reasons, courses offered using Java tend to be very popular with students, even with those who are not computer science majors. The size of this audience is hard to ignore. If presenting several introductory courses isn't feasible, can the goals of CS 1 be met using Java or maybe Visual Basic? Or, for that matter, should the selection of the programming language be a prime consideration?

At Brooklyn College and other schools it is now common to include the teaching of some HTML and Java Scripts in the breadth first course. This illustrates that language choices have become somewhat market and demand driven -- rather than purely based on academic issues. Furthermore HTML is a relatively simple language to learn and use.

From a historical perspective it appears that very satisfactory courses have been developed with a wide variety of programming languages. The determining factor for which way the pendulum swings in terms of a language's adoption for CS1 may be image and marketing factors, which certainly cannot be ignored. If an introductory course is carefully designed to meet its desired outcomes, it may not matter how we elect to express our algorithms. Although this may not be easy to accept, it may to be perfectly defensible to use whatever language the market dictates to ply our trade. Academic integrity need not be compromised by pursuing this rationale.

CS1 IN BREADTH FIRST CURRICULUM

Since the introduction of the ACM 1991 Curriculum (Tucker et. al, 1991) advocating a breadth-first approach as part of a reorganization of the entire computer science undergraduate curriculum, we have gained considerable experience using a number of approaches. The fundamental idea behind the breadth-first curriculum is that computer science, like the other physical sciences and biological sciences, has a core of topic areas (including Algorithms, Programming Languages, Architecture , Software

Engineering, Databases, Operating Systems, Artificial Intelligence, and Social Implications) which need to be introduced in the first course and then developed throughout the curriculum. In order to be accepted as a "hard" science, computer science also needs to have labs, with theoretical underpinnings, tests of practical environments with experimentation, and results which can be analyzed. Here we would like to share our experiences and consider their implications, while reflecting upon their effectiveness.

Each semester some 120 first year cadets at the United States Coast Guard Academy (USCGA) in Connecticut were required to take the introductory Foundations of Computer Science Course. Invariably an approach to teaching a course will be tied to the methodology of a selected text for a course. An early approach, used between 1991 and 1993, involved two texts, *Karel the Robot* (Pattis, 1994) and *Great Ideas in Computer Science* (Biermann, 1990). We found this an intriguing combination of texts to serve the function of a breadth-first introductory course. Karel introduced, developed and tested the methods of top-down design and step-wise refinement. For this purpose it served better than any software or textbook employed in our many years of experience in trying to teach the method and its importance. Students were able to develop hands-on experience with modular program design just by working with the *Karel* software. The effectiveness of Karel could to some degree be attributed to the power of the abstraction it portrayed: a robot/soldier (akin to first-year cadet students) with very specific tasks and the instructions to accomplish them. The robot needed to be obedient and disciplined, while its rules and goals of movement were well-defined. In certain situations there were restrictions on the robot's movements (implemented as conditionals) and in other situations the movements could be repeated (implemented as loops).

The Biermann text served excellently to cover a broad range of topics in computer science. Typically topics covered early in this approach included decision design, trees, text manipulation and algorithm design, functions, top-down programming, arrays, recursion, software engineering, and electric circuits. As a final addendum, topics like non-computability or artificial intelligence could be added. Overall, this produced a quite demanding set of topics for an introductory course required for non-majors. There were three-hour closed labs for this course using Karel and then Think Pascal 4.0 on a Macintosh. The instruction of programming comprised about 65% of the course. One main criticism of the Biermann text is its very poor and small, colorless, print size – but here the publisher, not the author is to blame.

Another approach used for the one year (two semesters) was *Computer Science: An Overview (4th ed.)* By Glenn Brookshear (1994). After a nice introduction to the history and evolution of computer science, this book dives right into the computer (almost literally) with Part One: Machine Architecture. As such an early stage in the course, as an introduction to a computer computer science curriculum, an approach which is too low level such as this can serve as a deterrent to the student considering the pursuit of computer science as a major subject.

As the Brookshear text moves further away from the details of machine architecture (Part One) it becomes more pleasant to use. If the instructor can concentrate on Part Two: Software, then coverage via the chapters Computer Systems and Network Algorithms, Programming Languages and Software Engineering is quite effective. Brookshear's exercises with instructor's solutions are also quite useful. Parts III (Data Organization with chapters on Data Structures, File Structures, and Database Structures) and IV (The Potential of Algorithmic Machines) is also quite useful. It is unlikely that one can get as far as Chapters 10 (Artificial Intelligence) and 11 (Theory of

Computers) without skipping a number of earlier chapters in a one-semester course. The Brookshear text came with a lab designed in C (now upgraded to C++ and Java) for three-hour closed labs. The labs did not blend well with the text (they were not directed related to the text). This approach in trying to teach programming by having students correct errors in syntax and logic simply does not work.

A most satisfying approach to delivering the breadth-first CS1 course were the text and methods of Schneider and Gersting, *An Invitation to Computer Science* (1995, 2nd Ed. 1998). This was used for a number of sections of the course (approximately 120 students each semester) taught to all incoming freshman. This approach essentially presents the whole subject of computer science in a top-down order, essentially reversed to that of Brookshear. It begins with problem-solving and algorithms, then goes into lower levels of the machine, including the hardware world (binary numbers, boolean logic, and gates), computer systems organization (Von Neumann architecture and historical perspective), the virtual machine (assembly language and operating systems), the software world (a very large segment of the text), followed by applications, and an excellent concluding section on social issues by guest contributor Sara Baase.

The text of Schneider and Gersting is so well integrated with the labs that the whole approach may be best remembered by the lab experiences. The labs are specifically designed for a hands-on approach with all the important text topics. Algorithms are presented in pseudo-code, can be executed in real time or step mode, and can be viewed in diverse formats including machine language. There are labs on sorting algorithms (also in pseudo-code), sort timing, Turing machines, assembly language programming, machine language, and Pascal (the second edition uses C++ instead). There is also a Scheme lab especially elegant in allowing students to quickly taste functional programming without getting bogged down by syntax. These labs have been so effectively and "tightly" prepared that we have even found them to be of benefit in upper level computer science courses. For example, in teaching algorithms students are encouraged to get a quick and easy "hands on" understanding of measuring timing, algorithmic complexity, and diverse sorting techniques by exploring the labs which accompany Schneider and Gersting's text. Another domain where the labs could be of benefit is in presenting the ideas of functional programming quickly and efficiently by introducing SCHEME for a programming languages or AI course.

Wilmington College in Ohio has employed both depth-first and breadth-first schemes for the Introduction to Computer Science course in the past eleven years. This CS1 course serves computer science, computer information systems, and secondary math education majors, with an occasional “stray” thrown in. The turning point to breadth-first was the release of the ACM/IEEE curriculum revision in 1991 (Tucker, et. al, 1991). The depth-first approach had never been particularly effective for introducing this wide range of students to the fundamentals of the computing discipline. But until the publication of the Schneider and Gersting text cited above, no truly satisfactory breadth-first text was available. Since its arrival in 1994, that book has been the fundamental text in the introductory course. The absolute stability of the accompanying lab software package should also be noted, as a further endorsement. In the years of its use at the College, it has never crashed or locked up, a truly remarkable feat for software in the hands of freshmen!

Wilmington has also adopted Biermann’s *Great Ideas in Computer Science* (1990) but for use throughout the computer science curricula rather than in a single course. Each course requires readings from the book appropriate to the discipline under study. The book becomes truly a part of the student’s total academic experience and is well-worn by graduation.

The primary teaching language has been Pascal for over a decade with only a single exception. The argument for Pascal made above mitigated in its favor. The negatives of C and all its derivatives were adjudged too great to warrant a change. The choice of Pascal and the continuing decision to remain with it reflect the faculty's belief that teaching the concepts of a high-level programming language is best achieved with a "tight" language. Of course, this is one of the criticisms of Pascal, but it is also one of its greatest strengths. The use of Delphi in some upper-division courses is underway now as an extension of the use of "plain" Pascal and a way to initiate students into the rapidly advancing world of rapid application development environments.

The only exception to the use of Pascal was a single use of Karel the Robot. For a variety of reasons, the experiment was unsuccessful. That does not, however, mean the experience of the Coast Guard Academy might not be repeatable at Wilmington College. It may be tried again in the future.

MISCONCEPTIONS ABOUT COMPUTER SCIENCE

Many years of teaching computer science (over 60 years between the authors) at a number of colleges and universities have convinced us that there is often a very serious misunderstanding amongst colleagues in other disciplines about what the subject of computer science is comprised of. This misunderstanding may be attributable, to some degree, to the accessibility and pervasiveness of computers and computer applications in many people's lives. Many academic colleagues perceive computer science as either just *programming* or use of *applications*, neither of which is of course correct. Hence the development of the breadth first curriculum and its accompanying breadth first CS1 course also serves to educate students with diverse backgrounds, interests and majors as well as other faculty members about the actual subject matter of our discipline. At a number of schools it is common to have several choices of concentration within the Computer Science major such as software development and information sciences. Other typical areas of concentration which might benefit from the background provided by a breadth-first CS1 are computer or electrical engineering or computer hardware, as well as for example a business major with a minor in computer science.

DISCUSSION, SUMMARY AND FUTURE

Many years of experience have taught us that there is no perfect language for CS1. The choice is often almost a matter of religious fervor, rather than any specific language-related feature(s). Pascal, while didactically perhaps the soundest language, became unfashionable and out of favor with those enamoured with demands of industry. C++ emerged in the late 80s and 90s, but with the explosion of interest in the World Wide Web, Java has taken over as the primary language and metaphor for contemporary computer programming. Finally computer science study has taken on many aspects of a practical nature which can attract new disciples with diverse ambitions.

The breadth first approach to teaching CS1 can satisfy the needs a of number of audiences, including majors and non-majors. An important factor which should affect the choice of whether to offer a breadth first approach is how robust the computer science curriculum is. A larger computer and information science department, with a broad spectrum of courses can easily afford to use the breadth first approach as there are likely to be a number of courses which might serve as a "supporting cast". The development of programming skills would be expected to be addressed by other courses.

In a smaller school with a limited of number of computer science course offerings, the breadth first approach may be attractive to a large audience who may miss the necessary skills to develop their programming ability.

As we enter the new Millennium it is pleasing to see that computer science courses are beginning to address practical issues as well as theoretical ones.

And so the discussions continue. The ACM/IEEE guidelines permit (even invite) this diversity of issues. However, in planning for the next generation of curricular standards, past experience should be thoughtfully considered.

REFERENCES

Abelson, H., Sussman G. and Sussman J. (1985). Structure and Interpretation of Computer Programs. MIT Press.

Biermann, A. (1990). *Great Ideas in Computer Science.* Mit Press.

Brookshear, G. (1994). *Computer Science: An Overview* (4th ed.). Addison-Wesley.

Pattis, R. E. (1994). *Karel The Robot: A Gentle Introduction to the Art of Programming* (2nd Ed.) (Revised by Jim Roberts and Mark Stehlik, both of Carnegie Mellon University).

Schneider, M.G, Gersting, J.L. (1995) An Invitation to Computer Science. West Publishing Company. with Laboratory Manual (Macintosh Version).by Lambert, K. , Whaley, T.,

Tucker, A. (ed.), Barnes, B., Aiken, R., Barker, K., Bruce, K., Cain, J., Conry, S., Engel, Gl, Epstein, R., Lidtke, D., Mulder, M., Rogers, J., Spafford, E., and Turner, A. (1991). *Computing Curricula 1991*, ACM/IEEE-CS Joint Curriculum Task Force, ACM Press and IEEE-CS Press*, New York.*

Tucker, A., *et al.* (1995). *Fundamentals of Computing I, C++ Edition.* McGraw-Hill.

Wegner, Peter, (1970) ACM Visiting Scientist Program, St. Bonaventure University,.

Acknowledgement

The authors would like to thank Dr. Noson Yanofsky for his useful comments.