ANOTHER APPROACH TO SERVICE COURSES

Dr. William Mitchell
Department of Computing Science
University of Evansville

Introduction

This paper discusses the issues surrounding service offerings by Computer Science departments and focuses specifically on the first programming course. The approach described by the author has been developed to serve business students who seek an introduction to programming, but it applies also to most non-majors. The popularity of computer applications in the various disciplines as well as the widely publicized vocational opportunities in data processing induce ever more students to try their hand at programming. The embarrassment of riches in enrollment, however, brings with it multiple problems of staffing, machine resources, and curricular balance. Less obviously it also brings the pressure for instant success in serving this new population and thereby avoiding the splintering of programming education among interested disciplines, as happened with statistics instruction. Various viewpoints on solutions to these problems have been published, but little understanding of the nature and goals of the students involved has been evidenced. What follows is an explanation of a student-oriented approach to service course instruction which has been instituted at the University of Evansville (Mitchell 78).

The Issues

Beyond the majors and minors currently flocking to computer related disciplines is a much larger group of students who seek merely brief exposure to computer applications or perhaps a useful degree of skill in applications programming. Most computer science departments initially funneled these students into the department's introductory course along with the majors. But as the discipline matured and the numbers increased, this practice has become unsatisfactory, and the problem of the first course has emerged.

Several schools have reported success with a modification of the unified first course approach which retains common lectures, but coordinates separate laboratories for each of various interest groups (Prather 78, Gibbs 77, Unger 76). The benefits of this approach seem to include large throughput and a degree of individualization in language use and applications. But the increasing diversity of students seeking programming skills has caused many to challenge the effectiveness of a unified first course. The growing numbers of students now make the possibility of discipline oriented courses economically feasible, and it is apparently for non-pedagogical reasons that Gibbs, Prather and Unger each take unified approaches as an implementation constraint. While there were once extensive philosophical arguments in favor of unified courses in college mathematics, no similar justification of a unified first course in computing science has been suggested.

What has been recommened as the content of the first course in programming varies significantly from those which assume mathematically sophisticated majors (Salton 73, Gries 74, Maly 75) to those which assume mathematically naive non-majors (Leitner 78, Cook 77, Taylor 77). Other authors have categorized computer users on a scale from those seeking "appreciation" to those concerned with "usage" (Adams 72, Solntseff 78). It has been argued that students in the usage category (practical programming experience is to be acquired) are broadly similiar as a consequence of their objective (Schneider 78). However, Addison Wesley documents the evolution of several content approaches relevant to usage, and concludes that there is a trend toward at least three distinct first courses (Gruener 78).

Growing experience emphasizes the necessity for a service orientation for some first courses. The non-major is a distinctly different student from the majors we commonly encounter. The University of Texas at Austin's attempt to correct the dilution of the unified introductory course resulted in a 25% drop rate (Chanon 77). The non-majors are not intellectually deficient, but their motivation to develop competence in programming applications is often weak and their preparation for the task is often meager. They are not programming because they are intrigued and fulfilled by the experience, but because they are required to gain exposure or they thought it would lead to a lucrative job.

Yet even special non-major courses are not overly successful. A study at Penn State to resolve the mutal dissatisfaction of business students and the CS faculty with a business Fortran course concluded that there was no obvious solution (Willoughby 73). Non-major courses capitalize on specialized interests, allow a more sedate pace, and permit the excising of more difficult, theoretical topics (Cook 77), but they have not been demonstrated to be significantly more effective in accomplishing the acquisition of useful programming skill.

One might conclude that despite the growing body of experience and experimentation, we have not yet grappled with the fundamental problems of the service course. The author suggests that for too long our students have self-selected themselves into computing science and we as faculty have never had to satisfy the needs of students who lack the common psychological and intellectual attributes we have learned to take for granted. As long as we continue to approach the students now enrolled in our service courses in the same way we approach our majors, we will continue to be dismayed by their lack of accomplishment.

A Student-Centered Analysis

Before you can deal successfully with the responsibilities of service instruction you must confront the question "Is programming for everyone?" The growing trend toward appreciation courses seems to reflect a judgment about programming paralleling the judgment arrived at in mathematics concerning calculus instruction. The evolution of the first course in computer science seems to retrace the evolution of liberal arts mathematics, discipline-oriented "applied" calculus, and "regular" calculus. The difference is the conviction by many that abstract mathematics is not at "the heart of computer science" (Gibbs 74) and the knowledge that junior colleges and trade schools have been producing competent programmers for years. The concrete nature of computer applications and their close modeling on common clerical procedures suggests that programming should be more accessible than calculus. Why then do so many students in our service courses have such great difficulty acquiring programming skill? What makes them so different in this regard from our majors? Ability is not the explanation, for the junior college does not attract the high ability student. The explanation more likely lies in the motivation and intellectual experience of the student.

It has been our observation over several years of teaching programming to predominantly business students in a general education course that they would eventually learn to write BASIC code which could be executed, but many never got beyond debugging at random. There never seemed to be a firm grasp of the process which was being executed by the machine. Many of those who did learn to use language well enough to complete their programming assignments successfully still required considerable direction in planning how to code simple applications. The course supposedly complied with the request of the business school to provide their students with actual programming experience sufficient to build an appreciation of the commercial programmer's task and the capabilities of the computer when applied to business problems. The students did not foresee practicing programming, but were being prepared to manage or interact with a corporate data processing function. The difficulty of dragging a class through expressions, loops, branching and arrays, however, usually left little time for considering character manipulation and file processing, even in a 4 quarter hour course. About all most students appreciated upon completion was that programming was an exceedingly detailed and complex task for which they had neither the time nor talent.

Our students, with few exceptions, have never before had occasion to use a computer. Some have no real interest in the experience and are only satisfying a requirement, but at least as many would like to know more about how computers are utilized in accounting, inventory, purchasing, and other business specialities. But programming was a much different activity than they were used to, and they had little in their background which related to it. They were asked to assimilate and integrate a massive collection of details and rules without a context. Programming was like working a jigsaw puzzle, trying to find the right pieces and fit them together so that they would make a coherent picture, but they were not puzzle solvers. As we examined our student's characteristics we found that they had avoided science and mathematics as much as possible, and with them the requirement to observe rigid rules and develop deductive reasoning. They were informal and imprecise in their thinking patterns and their manner of expression. They had had little opportunity to develop skills in detailed planning and foresight, or in discovering patterns and relationships.

Before coding and mechanical execution of programs can be made meaningful to these students, they must be provided with a context which will organize and prioritize the details. Before these students can appreciate the application of the computer as a problem solving tool, they need an understanding of the problem solving process and of procedures. It is fashionable to begin most introductory courses and text books with a section on algorithms, but this is actually intended only as a formalization of concepts intuitively understood. Most of these students lack this intuition. A single lecture of definition and examples is not a context on which to pursue the development and application of algorithmic thinking. Algorithms are already well down the tree of problem solving strategies, and the majority of these students are not conscious of any formal methodologies for specifying problems and describing their solutions.

At the University of Evansville we have designed a short course in problem solving and algorithms into which we direct any non-major student who wishes to pursue a first experience in programming. This course intends to make students aware of problem solving methodologies and to focus their attention on procedural thinking. Its style and content incorporates the content of Cashman and Mein's problem solving module (Cashman 75) but preceeds it with exposure to general problem solving (James Adams, Conceptual Blockbusting, Freeman, 1976) and extends it with a more detailed consideration of algorithm representation and analysis. Turing machines, flowcharting, Nassi-Schneiderman diagrams, decision tables, top-down decomposition, and procedural validation and testing are presented and illustrated with generally non-numerical examples. In effect, the original 4 quarter hour course is now divided so that 2 hours are devoted to the derivation, specification and testing of procedures without ever encountering a computer. Most students take this course and the following 2 quarter hour course in BASIC coding in sequence, so that the separation of programming concepts and language instruction advocated by

Fisher, Hankley and Wallentine is strongly enforced (Fisher 73).

We derive several advantages from the external division of these topics into separate two quarter hour courses. Separate courses focus both the student's and the instructor's attention on the topic at hand. When working code is the ultimate goal in a course it is impossible to prevent the student from focusing on coding. In the first course the instructor is forced to deal with the real weakness of the student and is not diverted by the duty to respond to premature questions about syntactical detail and coding techniques. The first course forces the student to recognize the role of conceptualization and planning and accustomizes him to thinking about procedures independent of a programming language. When the ultimate goal of the course is the ability to devise and represent an algorithm clearly, students strive to accomplish it.

Our experience with the two-course sequence is that more is learned which is useful to the student. In the first course he learns to tackle problems by decomposition, to represent the algorithms in a structured manner, and to test the complete procedure in its logical form before ever worrying about its implementation. In the second course we are able to deal with more coding and applications because the concepts of loop, decision, array and file have been previously introduced and exercised. In the first course we sketch the logic of various business applications, and in the second we cover the coding techniques and the language facilities for implementing them. The relation of programming language to algorithm remains clear and well ordered. The opportunity to confuse code with thought, to let the computer fill in the details or find the errors, is denied.

The division into separate courses permits the use of graduate students as instructors for the second course, where the goal is the explanation and exercise of the features of our interactive BASIC system, and allows the senior faculty to confront the intellectual difficulties posed by procedural thinking. Students are asked to challenge fewer concepts at a time and given more time for the concepts to mature before they have to test them on the machine. The cost of failure is less and the reason for failure is easier to diagnose. On the other hand, students with better backgrounds or previous computing experience may skip the first course and proceed to a fast-paced language course which is able to pursue nontrivial applications.

Finally, we believe that a course in problem solving and algorithmic thinking is a viable general education offering in its own right. It is offered with no mathematical prerequisite beyond first year high school algebra so it is more accessible than the course popularized by Rubinstein at UCLA (Rubinstein 75). While it teaches no language, it provides significant insight into the nature of computer solutions, and as such, is a valid introduction to computers which can be paired with our two quarter hour computer and society course to meet the needs of departments such as sociology and nursing.

We wish we could expand each of these courses into full four quarter hour offerings, but we have been constrained by the School of Business to provide meaningful programming experience within a four hour package. The advantages of the current sequence for the business student are many, and most are also appreciated by other non-majors. He now actually gets to learn some of the programming techniques utilized in commercial data processing and can appreciate the more fully the utility of the computer. He is able to consider more complex problems which are not only more meaningful but require him to bring to bear more of the programming language's characteristics. He learns that methodical problem solving and algorithms are applicable beyond programming, and that the distinction between coding and analysis is a valid one. He acquires through the problem solving course a context for tackling problems, an understanding of procedures, and techniques for specifying detailed plans. This context makes the details involved in computer interaction comprehensible, and conveys a clearer understanding of the machine's contribution to the problem solution.

The student who completes this introductory programming sequence is welcomed into the other language courses offered by the department (COBOL, RPG, FORTRAN, PL/I, etc., all taught as second languages) and other courses required of our majors, such as introductory hardware and introductory systems analysis. But the introductory programming course offered to our majors is quite different, for they come with different attitudes, different preparation and the need of different skills. Even so, weaker majors often elect to take the problem solving course after they have difficulty with that aspect of their first course.

SUMMARY

Arguments to include more instruction in problem solving and algorithms in the introductory course have been repeatedly offered in the literature, but invariably this has been addressed toward better education of the majors. The author argues that a service course aiming at developing programming skills in groups of non-majors must recognize that these students will lack basic concepts of procedural thinking and algorithmic problem solving which are taken for granted in majors. Without first supplying these concepts, the efforts devoted to elaborating programming techniques and language details will be largely wasted. It is the inability of these students to structure problems in procedural ways which inhibits their growth as programmers even as it interferes with their acquisition of language syntax. The adoption of a very basic problem solving course as a prerequisite to an introductory coding course at the University of Evansville was explained, and the benefits attributed to this approach were described.

REFERENCES

(Adams 1972)
Adams, J. M. and D. H. Haden, "Introductory Service Courses in the Computer Science Curriculum," SIGCSE Bulletin, Vol. 4, No. 1, pp. 49-52, March 1972.

(Cashman 75)
Cashman, W. F., and W. J. Mein, "On The Need For Teaching Problem-Solving In A Computer Science Curriculum," SIGCSE Bulletin, Vol. 7, No. 1, pp. 40-46, February 1975.

(Chanon 77)
Chanon, R. N., "An Experiment with an Introductory Course in Computer Science," SIGCSE Bulletin, Vol. 9, No. 3, pp. 39-42, August 1977.

(Cook 77)
Cook, Robert N., "An Approach To The Introductory Computer Science Course for Non-Majors," SIGCSE Bulletin, Vol. 9, No. 3, pp. 30-33, August 1977.

(Epley 78)
Epley, Donald and Ted Sjoerdsma, "A Two-semester Course Sequence in Introductory Programming Using PL/I--A Rationale and Overview," SIGCSE Bulletin, Vol. 10, No. 3, pp.113-119, August 1978.

(Gibbs 74)
Gibbs, Norman, B. Loveland, and T. Orkga, "The Heart of Computer Science", SIGCSE Bulletin, Vol. 6, pp. 13-44, December 1974.

(Gibbs 77)
Gibbs, Norman E., "An Introductory Computer Science Course for all Majors," SIGCSE Bulletin, Vol. 9, No. 3, pp. 34-38, August 1977.

(Gries 74)
Gries, David, "What Should We Teach In An Introductory Programming Course?" SIGCSE Bulletin, Vol. 6, No. 1, pp. 81-89, February 1974.

(Gruener 78)
Gruener, William B. and Steven M. Graziano, "A Study of The First Course In Computers," SIGCSE Bulletin, Vol. 10, No. 3, pp. 100-107, August 1978.

(Fisher 73)
Fisher, P., W. Hankley, and W. Wallentine, "Separation of Introductory Programming and Language Instruction," SIGCSE Bulletin, Bol. 5, No. 1, pp. 9-14, February 1973.

(Leitner 78)
Leitner, Henry and Harry R. Lewis, "Why Johnny Can't Program, A Progress Report," SIGCSE Bulletin, Vol. 10, No. 1, pp. 266-276, February 1978.

(Maly 75)
Maly, Kurt and Allan Hanson, "A First Course in Computer Science: What It Should Be and Why," SIGCSE Bulletin, Vol. 7, No. 1, pp. 95-101, February 1975.

(Mitchell 78)
Mitchell, William and Bruce Mabis, "Implementing a Computer Science Curriculum Merging Two Models," SIGCSE Bulletin, Vol. 10, No. 3, pp. 151-155, August 1978.

(Prather 78)
Prather, Ronald and Judith Schlesinger, "A Lecture/Laboratory Approach to the First Course in Programming," SIGCSE Bulletin, Vol. 10, No. 1, pp. 115-118, February 1978.

(Rubinstein 75)
Rubinstein, Moshe, PATTERNS OF PROBLEM SOLVING, Prentice Hall, Inc. 1975.

(Salton 73)
Salton, Gerard, "Introductory Programming at Cornell," SIGCSE Bulletin, Vol. 5, No. 1, pp. 18-20, February 1973.

(Schneider 78)
Schneider, G. Michael, "The Introductory Programming Course in Computer Science Ten Principles," SIGCSE Bulletin, Vol.10, No. 1, pp. 107-114, February 1978.

(Smith 76)
Smith, C. and J. Rickman, "Selecting Languages for Pedagogical Tools in the Computer Science Curriculum," SIGCSE Bulletin, Vol. 8, No. 3 pp. 39-47, September 1976.

(Solntseff 78)
Solntseff, N., "Programming Languages for Introductory Computing Courses--A Position Paper," SIGCSE Bulletin, Vol. 10, No. 1, pp. 119-124, February 1978.

(Stokes 74)
Stokes, Gordon E., "Service Course Position Paper," SIGCSE Bulletin, Vol. 6, No. 3, pp. 18-22, September 1974.

(Taylor 77)
Taylor, Robert P., "Teaching Programming to Beginners," SIGCSE Bulletin, Vol. 9, No. 1, pp. 88-92, February 1977.

(Unger 76)
Unger, E. A. and N. Ahmed, "A Instructionally Acceptable Cost Effective Approach To a General Introductory Course," SIGCSE Bulletin, Vol. 8, No. 2, pp. 28-31, June 1976.

(Willoughby 73)
Willoughby, Theodore, "Student Attitudes Toward Computers," SIGCSE Bulletin, Vol. 5, No. 1, pp. 145-147, February 1973.