

IS THERE A ROLE FOR PROGRAMMING IN NON-MAJOR COMPUTER SCIENCE COURSES?

Mark Urban-Lurain¹ and Donald J. Weinshank²

Abstract - *Should non-CS majors learn to program? While the 1999 National Academy of Sciences report *Being Fluent with Information Technology* [8] advocates teaching programming as part of the CS-0 experience, we challenge the assumptions upon which this recommendation rests. Our extensive review of the NECC and SIGCSE conference proceedings from 1979 to 1998 clearly shows a decline in the number of articles in which programming is taught in CS-0 courses. Furthermore, based upon learning theory literature, we argue that conceptual understanding of computing can be acquired without learning to program. Finally, we describe our criterion-referenced, mastery-model course that prepares 1800 students per semester for a computing future that is constantly changing.*

INTRODUCTION: FITNESS

Should non-computer science (non-CS) majors learn to program? The 1999 National Academy of Sciences report *Being Fluent with Information Technology* [8] advocates that all college students should learn to program to ensure that they are “fluent with information technology (FIT).” The report claims that the term “computer literacy” has come to be associated with superficial training on some computer applications rather than concentrating on deeper conceptual understanding that will prepare students to cope with rapidly changing information technology. FITness requires three types of knowledge:

- Contemporary skills, the ability to use various computer applications.
- Foundational concepts, the basic principles and concepts of computing that form the basis of computer science.
- Intellectual capabilities, the ability to apply information technology in particular situations and use this technology to solve new problems.

The report acknowledges that the role of programming in understanding information technology has been the topic of many debates among computer scientists.

When we investigated what computer science departments have actually been doing to address the needs of non-major students, we identified several trends at variance with the FIT report. We describe these trends in our review of the ACM literature on non-major CS courses. We then address the question of programming in the context of learning theory. Finally, we

summarize the approach we have created for teaching non-computer science majors at Michigan State University.

ACM LITERATURE REVIEW

The Association for Computing Machinery (ACM) is the oldest professional computing organization. ACM publishes recommendations for CS undergraduate curricula. [1, 2] However, they have not published guidelines for CS-0 “service courses” for non-computer science students.

Within ACM, the Special Interest Group on Computer Science Education (SIGCSE) provides a forum for university faculties who are involved in teaching computer science to exchange information. They hold annual conferences and publish regular bulletins and conference proceedings [3]. Until 1997, ACM SIGCSE was associated with the National Educational Computing Conference but ended that association because the foci of the two organizations were diverging, with NECC concentrating on computing in K-12 and SIGCSE focusing on higher education. These publications contain articles about teaching computer science, including CS-0 courses.

To determine how CS departments have addressed the CS-0 course, we examined all of the National Educational Computing Conference Proceedings from 1979 through 1981, and all of the SIGCSE Bulletins and SIGCSE Conference Proceedings from 1982 through 1998, vol. 1, (the annual SIGCSE Conference Proceedings.) We selected all articles in which the primary focus was CS-0 courses. These articles represent what departments are actually doing in their CS-0 courses, as opposed to what computer science faculties believe should be done.

As we reviewed these 54 articles, we classified each into one of seven categories based on its primary emphasis. In alphabetical order, the categories are:

- **Applications.** These are courses in which the primary instructional focus is on learning about computers through the use of a variety of application software.
- **Management.** In recent years, the demand for CS-0 courses has significantly increased enrollments and course sizes at many institutions. As a result, some articles focus on the logistics and management of courses of this size.
- **Multimedia.** The primary instructional focus of these courses is in the creation of multimedia content.

¹ Mark Urban-Lurain, Michigan State University, Department of Computer Science and Engineering, E. Lansing, MI 48824, urban@cse.msu.edu

² Donald J. Weinshank, Michigan State University, Department of Computer Science and Engineering, E. Lansing, MI 48824, weinshan@cse.msu.edu

- **Programming.** The primary focus of these courses is on learning to program in a variety of different languages including Basic, Scheme, Pascal, and others.
- **Simulations.** The emphasis here is on learning about computers by creating or using simulations.
- **Social.** The course emphasis is on the social impact of computers and their roles in a changing society.
- **Survey.** This category includes courses that are intended to be an introduction or overview to the discipline of computer science and "computer appreciation" courses.

The complete list of articles and their classifications may be obtained at <http://aral.cse.msu.edu/Publications/FIE2000/ProgrammingInCS-0.html> The distribution of the various types of articles over the years is shown in Figure 1.

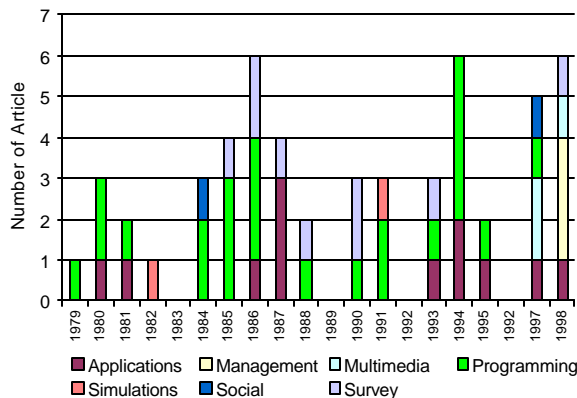


FIGURE 1
DISTRIBUTION OF CS 0 ARTICLE TYPES BY YEAR

DISCUSSION

The conventional wisdom among CS faculty has been that learning programming is the first and most fundamental way to learn about computers. Hence, programming has been the major focus of most courses until very recently. In the decade between 1979 and 1988, 52% of the articles had a programming focus. However, the number of courses with programming as the primary emphasis has decreased recently. From 1990 to 1998, 36% of the articles had a programming focus, but there was a downward trend. By 1998, there were no articles about CS-0 courses in which the primary focus was programming. In that year, the emphasis was on managing the demands of large student enrollments in these courses. We believe this shift is due to institutional requirements such as the increased heterogeneity of the student body and the increase in the number of colleges and universities now requiring technological competence among students in a wide variety of disciplines.

Most computer science departments do not include the CS-0 course in their curriculum concerns. Such courses are often taught on a rotation basis, with each faculty member doing whatever s/he deems appropriate. However, many in higher education are

concerned about the role of technological expertise and understanding as a core part of educating students. Computer science faculties are increasingly being called upon to define and teach FITness.

TEACHING NON-MAJORS TO PROGRAM DOES NOT TEACH TRANSFERABLE CONCEPTS

Soloway [19] asked well-known computer scientists to discuss the need to learn programming. Soloway uses the metaphor of programming as the "new Latin." He notes that learning to program is claimed to have benefits ranging from teaching general-purpose problem solving and thinking skills to helping students appreciate and understand how computers work. This is much like the claim that learning Latin helps general cognitive development, leading to better learning in other domains. Soloway claims that teaching programming to non-CS majors is no longer necessary to be a sophisticated computer user. His colleagues disagree and assert that learning to program does indeed have general educational benefits.

Claims that learning computer programming transfers to other domains and improves general problem solving are widespread. Mayer, Dyck and Vilberg [13] reviewed several studies that examined the relationship between learning to program and general problem solving abilities. They point out that the few studies that do support the claim that learning to program enhances general thinking abilities are either based on anecdotal or personal introspection data, both of which are unreliable. They conclude that "there is no convincing evidence that learning to program enhances students' general intellectual ability, or that programming is any more successful than Latin for teaching 'proper habits of mind'" (p. 121).

In another test of transfer to new domains, Kurland, Pea, Clement and Mawby [11] studied the impact of learning programming on the general problem solving skills of high school students who had studied programming for two years. The authors found that not only did the students general problem solving not improve, the students actually had little understanding of programming at the end of two years of study. The authors conclude that the pedagogy of teaching programming needs further study before it is possible to begin making claims that learning to program transfers to other domains.

It is a fallacy to claim that programming is "mental calisthenics" that builds strong minds: there is a fundamental difference between expert and novice knowledge. Experts have a large repertoire of knowledge from which to draw, and their knowledge is "conditionalized." [17] That is, experts have the broad range of experience and knowledge to understand the appropriate conditions under which to apply particular principles. For example, this ability allows chess experts to consider only a subset of moves, rather than performing an exhaustive search of all possible moves.

This principle also holds true in the domain of programming. Lewis and Olson [12] point out that expert programmers have several sets of "operations" (collections of sequential statements

that perform common programming functions) that they can apply to classes of common programming problems when they encounter them. Novice programmers approach each of these classes of problems as if they are unique and must grapple with them as new problems rather than being able to apply known solutions to a recognized classes of problems.

Holt, Boehm-Davis and Schultz [10] compared expert programmers' cognitive representations of software with those of college student programmers. Both groups were asked to modify programs written by other programmers and then to describe their mental models of the programs. The experts' mental models were influenced primarily by the difficulty of the modifications they had to perform on the programs. This group categorized the programs based on the similarities and differences in the types of modifications, invoking sequences of actions that form sets of operations [12]. In contrast, the novice programmers' mental models focused on the structure and content of the programs. This group tended to categorize the programs based on the similarities to other such programs they had encountered (e.g., programs that use search trees, linked lists, etc.), rather than on the modifications requested.

Expert knowledge allows experts (e.g., computer scientists) to see the connections between programming and problem solving in other domains. They are doing what all learners do as they construct knowledge: attempting to connect that which they do not know to that which they do know.

However, concluding that programming causes improved problem solving in other domains is the classic logical fallacy of *post hoc, ergo propter hoc*. It is the deep expert knowledge, not the mere exposure to programming, which improves problem solving in other domains. This type of transfer requires expert knowledge acquired only after years of study. It is precisely this expert knowledge base that the novice not only lacks but also cannot learn in one or two courses.

THE IMPORTANCE OF CONCEPTUAL UNDERSTANDING

We agree with the National Academy of Sciences report's framework that FITness requires three dimensions: intellectual capabilities, conceptual knowledge and an appropriate skill set. It is only through conceptual understanding that individuals can transfer their knowledge to create solutions to unique problems and adapt to rapidly changing information technology. However, we challenge the report's recommendation that the way to obtain the conceptual understanding is to learn to program. Programming was originally necessary to be FIT because all interactions with the computer required programming. Interaction with information technology has now moved to higher levels of abstraction. For example, where once collecting and analyzing data required writing computer programs, today we use spreadsheets to do these tasks with much greater ease and sophistication.

If students do not program, how are they to learn and understand the computing concepts? Will they simply learn a set

of isolated skills that will be obsolete as soon as they leave the class? Our experience was that both the skills-only and the programming-plus-applications approaches [24, 25] failed. Students arrived in subsequent courses unable to perform all but the most rudimentary of tasks. They had not retained what they had learned with either approach and could not transfer their knowledge to comparable problems. Students who had learned particular keystrokes in specific applications software did not have the conceptual understanding to segue into new software environments. On the other hand, students who had learned programming could not see the applicability of these constructs to the problems they were trying to solve. This is because programming constructs (e.g., control structures, variables) are too far removed from the concepts (e.g., hierarchical file systems, data representation) that are needed for transfer among application software.

COMPUTING CONCEPTS AND COMPETENCIES AT MICHIGAN STATE UNIVERSITY

To address these issues, we implemented a large (1,800 students per semester) course during fall 1997 [21-23]. This course was designed based on extensive needs-assessment interviews with the chairs of the 67 departments whose students take the course and with the academic deans of their colleges. To ensure that the course meets the design goals and stays current with the changing demands of the client departments, we meet semi-annually with an oversight committee consisting of the academic deans of the client colleges. We review the course content, student performance, changing computing environments, the success of students who have taken the course in their subsequent courses and changing client needs.

Rather than taking a deductive approach in this course, we adopted an inductive approach. We introduce students to computing concepts by having them solve a series of problems that epitomize classes of problems for which various computing skills are the solutions. [15] For example, students encounter the concept of "data representation" in the HTML of their Web sites, in collections of abstract formatting attributes in word processing documents and in the relationships between numeric and graphic representations of data in spreadsheets. In each case, students learn particular skills, but the focus is on how those skills are examples of the underlying ways in which computers represent data.

We created a spiral curriculum [6] in which students have to solve increasingly challenging problems using a variety of software packages. While solving the particular problems, students construct mental models or schemata of the computing systems they are using. They thus build from procedural skills towards conceptual understanding, rather than first trying to learn decontextualized concepts and then attempting to use those concepts to solve problems. [20, p 45] As students grapple with apparently unrelated problems, the instructor ties them together, showing how each is an example of particular concepts or principles. Subsequent instruction relates the new problems to the previously learned concepts and principles. Students thereby "triangulate" on these concepts and principles, refining their

schemata as they solve successively more abstract problems. This approach is consistent with a constructivist perspective on how novice knowledge evolves into expert knowledge [18, p 148]. Figure 2 represents the process.

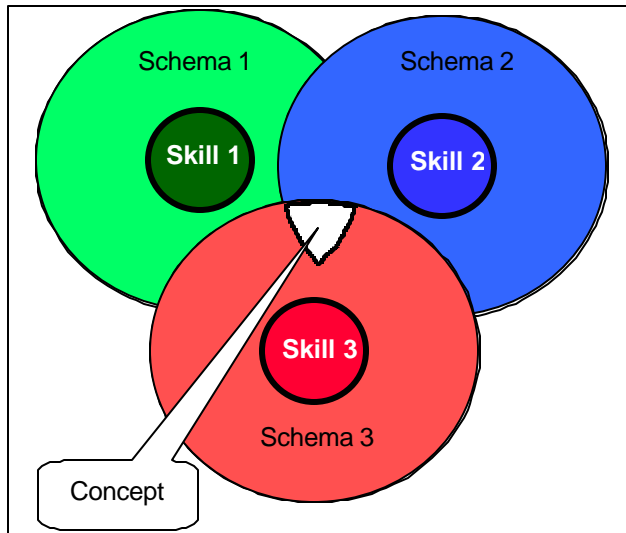


FIGURE 2
SKILLS, SCHEMA AND CONCEPTS

Students learn Skill 1 (represented by the smaller circle) and build a mental schema representing a generalization of this particular skill (indicated by the larger circle), Schema 1. This schema may consist of accurate representations of the computing concepts along with misconceptions. Students then learn subsequent skills and build corresponding schemata. The intersection of these schemata triangulates on a more accurate representation of the concept. [4, 16] Understanding the concepts underlying an increasing number of skills reduces the cognitive load required to represent the knowledge, as compared with the requirements of storing an increasing number of discrete, unrelated skills [7]. Ultimately, the students' conceptual understanding becomes rich enough to support independent problem solving beyond that which is possible with solely procedural skills. Although this approach entails additional instructional expense over a more behavioral instructional model that focuses merely on skills [9], the goals of retention and transfer justify the additional effort.

HOW DO WE MEASURE SUCCESS?

With traditional assessments such as quizzes and multiple-choice exams, students' primary motivation is the extrinsic reward of "points." Little retention and even less transfer to new problems occurs. To address these issues, we created assessments – called *bridge tasks* (BTs) – that are authentic, hands-on tasks requiring the students to solve problems similar to those they will encounter in their major courses and the workplace [18, p 149]. Bridge tasks are criterion-referenced, modified mastery-model assessments [5, 14] designed specifically to test students' conceptual understanding and transfer. They incorporate "extension tasks" in which students

must solve new types of problems that cannot be solved by rote memorization of skills or procedures but instead require that they apply their conceptual understanding to problems that they have not previously encountered. More information and sample bridge tasks may be found on the course Web site at www.cse.msu.edu/~cse101 on the student Web pages.

Bridge tasks are evaluated on a mastery pass/fail basis. If a student passes the first bridge task, he or she "locks in" a minimum grade of 1.0 in the course. For each subsequent BT passed, the student's course grade is incremented by 0.5 until she or he has passed the 3.0 BT. (Once a student passes the 3.0 BT, s/he may do an integrative semester project that may increase the course grade to 3.5 or 4.0.) In the strict sense, this is a modified mastery model in that students must complete the five BTs within the confines of a semester (about 12 BT attempts), rather than having an unlimited amount of time. As in any mastery model, however, the course grade is based on individual student effort and accomplishment rather than the class average.

The feedback from the faculty in our client colleges and departments and from the academic deans at our semi-annual oversight committee meetings has been unanimous: we are meeting our goals of student retention and transfer. This result is in complete contrast with our previous courses in which transfer was nearly non-existent. When students arrive in their subsequent courses in their majors, they are now able to use computers successfully to solve new problems in a variety of domains across the university curriculum.

CONCLUSION

As non-CS students from a variety of disciplines prepare to be the "information workers" of tomorrow, they must be able to use a variety of rapidly changing computing systems and tools to solve an ever-expanding range of problems across disciplines. In our view, rapid advances in applications software and hardware platforms now require mastery of a set of concepts and skills far removed from programming. Instruction designed for this purpose must motivate the students' interest and engage them in authentic tasks so that they construct rich schemata and concepts of computing systems. FIT computer users must apply these concepts in constantly changing computing environments.

REFERENCES

A complete listing of the CS-0 articles we reviewed may be obtained at:

<http://aral.cse.msu.edu/Publications/FIE2000/ProgramminGInCS-0.html>

- [1]. ACM Curriculum Committee on Computer Science, "Curriculum 68: Recommendations for the undergraduate program in computer science", *Communications of the ACM*, Vol. 11, No. 3, 1968, pp. 151-197.
- [2]. ACM Curriculum Committee on Computer Science, "Curriculum 78: Recommendations for the undergraduate

- program in computer science", *Communications of the ACM*, Vol. 22, No. 3, 1979, pp. 147-166.
- [3]. ACM Special Interest Group on Computer Science Education, "ACM SIG fact sheet", http://www.acm.org/sigcse/sigcse_fact_sheet.html, 1998,
- [4]. Anderson R.C., The architecture of cognition, Harvard University Press, Cambridge, MA, 1984.
- [5]. Block J.H., Eftim H.E., Burns R.B., Building effective mastery learning schools, Longman, New York, 1989.
- [6]. Bruner J.S., The process of education, Vintage Books, New York, 1960.
- [7]. Chandler P., Sweller J., "Cognitive load theory and the format of instruction", *Cognition and Instruction*, Vol. 8, No. 4, 1991, pp. 292-332.
- [8]. Committee on Information Technology Literacy, "Being fluent with information technology", National Academy of Sciences, Washington, DC, 1999,
- [9]. Foshay R., "Sharpen up your schemata", *Data Training*, No. May, 1991, pp. 18-25.
- [10]. Holt R.W., Boehm-Davis D.A., Schultz A.C., "Mental representations of programs for student and professional programmers", In: Olson GM, Sheppard S, Soloway E (eds) Empirical studies of programmers: Second workshop, Ablex Publishing Corporation, Norwood, NJ, 1987, pp. 33-46.
- [11]. Kurland D.M., Pea R.D., Clement C., Mawby R., "A study of the development of programming ability and thinking skills in high school students", In: Soloway E, Spohrer J (eds) Studying the Novice Programmer, Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1989, pp. 83-112.
- [12]. Lewis C., Olson G., "Can principles of cognition lower the barriers to programming?", In: Olson GM, Sheppard S, Soloway E (eds) Empirical studies of programmers: Second workshop, Ablex Publishing Corporation, Norwood, NJ, 1987, pp. 248-263.
- [13]. Mayer R.E., Dyck J.L., Vilberg W., "Learning to program and learning to think: what's the connection?", *Communications of the ACM*, Vol. 29, No. 7, 1986, pp. 605-610.
- [14]. Osin L., Lesgold A., "A proposal for the reengineering of the educational system", *Review of Educational Research*, Vol. 66, No. 4, 1996, pp. 621-656.
- [15]. Reigeluth C.M., Stein F.S., "The elaboration theory of instruction", In: Reigeluth CM (ed) Instructional-design theories and models: An overview of their Current Status, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983, pp. 338-381.
- [16]. Shuell T., J., "Cognitive conceptions of learning", *Review of Educational Research*, Vol. 56, No. 4, 1986, pp. 411-436.
- [17]. Simon H.A., "Problem solving and education", In: Tuma DT, Reif R (eds) Problem solving and education: Issues in teaching and research, Lawrence Erlbaum, Hillsdale, NJ, 1980, pp. 81-96.
- [18]. Smith J.P., III, diSessa A.A., Roschelle J., "Misconceptions reconceived: A constructivist analysis of knowledge in transition", *The Journal of the Learning Sciences*, Vol. 3, No. 2, 1993, pp. 115-163.
- [19]. Soloway E., "Should we teach students to program?", *Communications of the ACM*, Vol. 36, No. 10, 1993, pp. 21-24.
- [20]. Tennyson R.D., Cocchiarella M.J., "An empirically based instructional design theory for teaching concepts", *Review of Educational Research*, Vol. 56, No. 1, 1986, pp. 40-71.
- [21]. Urban-Lurain M., Weinshank D.J., "'I Do and I Understand:" Mastery model learning for a large non-major course" Special Interest Group on Computer Science Education, ACM Press, New Orleans, LA, 1999, pp. 150-154.
- [22]. Urban-Lurain M., Weinshank D.J., "Mastering computing technology: A new approach for non-computer science majors", <http://aral.cse.msu.edu/Publications/AERA99/MasteringComputing.html>, 1999,
- [23]. Urban-Lurain M., Weinshank D.J., "Computing concepts and competencies", In: Brown D (ed) Interactive learning: Vignettes from America's most wired campuses, Anker Publishing Company, Bolton, MA, 2000, pp. 73-74.
- [24]. Weinshank D.J., Urban-Lurain M., Danieli T., McCuaig G., Integrated Introduction to Computing, Kendall/Hunt Publishing Company, Dubuque, Iowa, 1992.
- [25]. Weinshank D.J., Urban-Lurain M., Danieli T., McCuaig G., Integrated Introduction to Computing, Kendall/Hunt Publishing Company, Dubuque, Iowa, 1995.