Behavior Research Methods, Instruments, & Computers
1987, 19 (2), 123-127

SESSION IV RESEARCH AND TEACHING AT COMPUTER-INTENSIVE CAMPUSES

Richard S. Lehman, *Chair*

The computer as a tool in instructional computing: Students as software architects

THOMAS T. HEWETT
Drexel University, Philadelphia, Pennsylvania

The computer-rich environment that exists when every student has a computer provides a variety of new opportunities for instructional use of computers, including new opportunities for the use of microcomputer application programs. After arguing that computer programming has a limited role in instructional computing, this paper describes the assumptions, development, and structure of a psychology course in which students make use of the microcomputer and its application programs as a tool in software design. However, programming is not required. Rather, the personal computer and its application programs provide an environment in which the student has the freedom to develop software design and explore course content without being constrained by the mind-numbing minutiae involved in programming a rigid, inflexible tutee.

One often hears of the introduction or use of computers in courses through the use of prewritten courseware packages (e.g., Castellan, 1983; Collyer, 1984), statistical analysis packages (e.g., MiniTab, SAS, SPSS^X, and BMDP), or programming exercises (e.g., Collyer, 1984; Taylor, 1980). There is, however, another alternative: application programs that convert the computer into a useful tool.

There are a variety of reasons for exploring the use of application programs in instructional computing. One reason is a result of the impact of the machine-rich environment created when every student has a personal computer. Just as the computer-rich environment forces a new perspective on available courseware (Hewett & Perkey, 1984) and on courseware development (Chute, 1986; Hewett, 1986c; Perkey, 1986), it forces a new perspective on instructional computing (Hewett, 1986c). Among the more salient features of the new instructional computing environment is the student's freedom of access to a computer. This ease of access creates a variety of new opportunities for instructional use of computers, including new opportunities for the use of microcomputer application programs (e.g., Hewett, 1985, 1986a, 1986b).

A second reason for exploring the use of application programs in instructional computing is based upon two assumptions about the use of programming exercises. The first is that programming is not generally useful in teaching or learning substantive course content outside of programming courses. The second is that when a student is asked to develop software for a content course, most of that student's mastery of content comes from the thinking that goes on behind program development, rather than from the actual implementation of the program. That is, it is in the design of the algorithm or of the architecture of the software that most of the learning takes place.

WHY PROGRAMMING IS NOT GENERALLY USEFUL

To understand why having students write programs appears to be useful, but is not particularly desirable in a content-oriented course, we need to understand why programming is sometimes thought to be valuable. Several authors have argued for the importance of having computer users learn to program, but most of the presumed benefits for students have been described by Taylor (1980) and by Collyer (1984). For example, Taylor distinguished between three "modes" of instructional use for computers: the computer as tutor, as tool, and as tutee. When the computer is functioning as tutor, it has been programmed by one or more "experts" in programming and in the subject matter of the tutorial. The computer becomes a tool when it has some useful application capability programmed into it. A good example of this is word

Requests for reprints should be addressed to the author at the Department of Psychology and Sociology, Drexel University, Philadelphia, PA 19104.

processing, which can be used in a variety of subjects. The computer assumes the role of tutee when the user must undertake to instruct the computer in the performance of some task, using a language the computer "understands."

The Benefits of Programming

Taylor (1980) argued that it is the role of the computer as tutee that offers the greatest long-range educational benefits. He gave three reasons for this conclusion. First, the human tutor must understand the task before he or she can teach it to the computer. Second, through work with computer logic, the human tutor will learn about the workings of computers and about the working of his or her own mind. Third, no time or money is spent on finding and/or acquiring predesigned tutor software.

Collyer (1984) also argued for the utility of programming as a part of instructional computing. Collyer suggested two major advantages in having students learn to program. The first of these advantages is that the student learns about the computer's intended task during the ongoing problem analysis that takes place while the programmer is trying to get the program to work. The second advantage is that students develop the sense of control and mastery that goes with successful accomplishment of a task.

The Problem with Programming

The advantage of programming suggested by both Taylor (1980) and Collyer (1984) is that a student can learn quite a lot about the task or course-content problem by trying to get the computer to accomplish it or solve it. We can grant the validity of this claim; that is not where the problem lies. However, an assessment of the worth of programming as a method of learning content materials does require some additional context.

Programming, in and of itself, is a useful exercise. It probably does have some of the benefits claimed for it as an aid in understanding the mathematical abstractions and concepts embodied in a programming language (Papert, 1980). Furthermore, it may even be a useful thing to do if one believes programming exercises and an understanding of computer logic generalize to other worthwhile areas of human endeavor. However, for many content areas, the argument that programming is an aid in understanding content is similar to the argument that using a machine to make a hammer is an aid in understanding how to use a hammer. It may be true, but it seems a rather cumbersome method for learning to use a hammer.

As anyone who has tried can attest, programming can be a laborious, awkward and time-consuming activity, requiring a lot of attention to details not relevant to solution of content problems. (For example, some programmers of my acquaintance claim that from 50% to 80% of the work required to implement a user-oriented application or tutorial program is associated with handling input/output requirements.) Furthermore, the level of programming sophistication required to work productively with substantive content problems in an upper level course, say cognitive psychology, is relatively high. In this case, a programming project can require a student to do extensive learning or relearning of programming skills. This learning may be fine if one is training programmers, but it can interfere with mastery of the psychology the student is supposed to be learning. Consequently, it is reasonable to wonder whether there are ways to achieve the presumed benefits of programming with less pain on the part of the student.

AN ALTERNATIVE TO PROGRAMMING

In a recent paper, Hewett (1986c) proposed that Brooks's (1975) architect-implementer model of software development could be applied to courseware development. Brooks argued for placing a premium on the conceptual integrity of software design. This concern with conceptual integrity led him to suggest that there should be a clear separation of roles in software development. Likening the design and completion of software to the construction of a building, Brooks proposed that a software development project should have both an architect and an implementer. (Somehow, "builder" seems more appropriate than "implementer.") The software architect, like the architect of a building, is to be the user's agent. "It is [the architect's] job to bring professional and technical knowledge to bear in the unalloyed interest of the user, as opposed to the interests of the salesman, the fabricator, etc." (Brooks, 1975, p. 45). In this case, however, the architecture Brooks is describing is "the complete and detailed specification of the user interface" (Brooks, 1975, p. 45). The builder, or programmer, is to have creative and inventive control over the actual implementation of the architect's design. Thus, the builder's job is to bring technical and construction skills to bear in developing a clean, realistic, efficient, smoothly operating product.

In the architect-builder model, the architect of a software project, who is charged with doing the conceptual design work, does not need, and is not expected, to do the programming required to make the software work. Programming is the role of the builder. One of the assumptions of the architect-builder model is that the architect's understanding derives from the conceptual design of the project, not from the programming itself. Another assumption of this model is that although knowledge about programming is important, programming by the software architect may be undesirable, especially if technical problems occupy the architect's attention to the point of interfering with realization of the architect's goals.

Adopting the architect-builder model for software development has some important consequences, not the least of which is the conclusion that programming is not essential to conceptual understanding and design. Furthermore, if, as suggested by Hewett (1986c), faculty do not have to become programmers to be courseware architects, then students do not have to become programmers to learn course content. The learning that takes place in trying to "teach" something to the computer can be accomplished in ways that do not impose the limitations of programming. Also, the sense of control and mastery over the machine that programming may provide can be developed in a single programming course or through the use of powerful application programs, such as the word processor or a spreadsheet. (With regard to the idea that working with computer logic enables a human tutor to learn about the working of his or her own mind, I must confess that I have no data, but it has not been my experience that programmers as a group possess any unique or special insights into the workings of their own minds.)

THE STUDENT AS SOFTWARE ARCHITECT

Some years ago, in response to both personal interests and long-range plans, I undertook to develop a course in software psychology—the application of experimental and cognitive psychology to an understanding of human-computer interaction. Starting off with the model found in Shneiderman (1980), I gradually modified the structure and content of the course to suit my own interests and conception of the field. Currently, the course focuses on two sets of issues—the cognitive psychology of computer programming and the application of cognitive psychology to the design and evaluation of human-computer interfaces, with greater emphasis on the latter topic. The 10-week course typically follows a mixed lecture, discussion, and demonstration format, with a midterm examination, a final examination, and a course project.

The Impact of Required Student Access to Computers

Originally, when there was no hardware to support the course, I developed a fairly traditional lecture course, covering the material without computers and without requiring programming. I had offered the course successfully a few times using a lecture-seminar format when the university announced a decision to require microcomputers for entering freshmen. During the spring quarter of 1982-83, I shifted the emphasis of the software psychology course from a literature survey to application design. While lectures on perception, learning and memory, and problem solving were scattered throughout the class meetings, the major focus of the course was upon having class members be part of a project design team. The problem posed to the students was this:

> Ignoring limitations of disk size, computer memory size, and programming difficulties, design the conceptual structure, and identify the content needed, in outline form, for an interactive disk. This must be done in the context of what we know about perception, learning and memory, and problem solving in humans. The goal of this disk is to train novice users of a personal computer and to provide them with all the information they need to be able to use the machine and the facilities at Drexel.

Most students in the course were graduating seniors majoring in computer science or computer systems management. All of the students had worked in private industry for more than one period of Drexel's cooperative education program. These co-op jobs typically afford Drexel students a great deal of hands-on work experience. Consequently, the class as a whole had a high level of work-related experience with a variety of computer systems under a variety of conditions. Furthermore, as one student observed at the end of the term, all of the students consistently "brought their brains to class with them."

One product of this class was a 12-page single-spaced topical outline of the information that is either necessary or useful in working with personal computers. An additional product was a 5-page set of design philosophy principles that made explicit the class's ideas about how to design interactive software. In the fall quarter of 1983-84, a second group of software psychology students, using the materials created by the earlier class, tackled the same problem and further refined the topical outline and design philosophy documents. Although only the content for an interactive disk was defined by these classes, their work did bear fruit, although it emerged in much different form than either class originally envisioned. An interactive disk—the Drexel Disk—was ultimately produced and distributed by the university. This information resource is a student-faculty guide to microcomputing and to the microcomputing facilities at Drexel (Hewett, Perkey, & Wozny, 1986; Wozny, 1986).

Impact of the Architect-Builder Model

It was these experiences that first showed me how much could be done by designing without programming, and the class has continued, albeit with a different focus, in a similar vein ever since. In addition to having come across Brooks's (1975) architect-builder metaphor, I realized that the number of Macintosh owners among my students had grown, and that several of them now had both machines and courses in which courseware and application software were used. In addition, those students who did not own personal computers could easily make use of one in the university-created access clusters. This meant that I could require individual students to do architectural design work for software projects.

Consequently, a major focus of the current version of the software psychology course is upon developing an appreciation of psychological principles through their application to the creation of a well-thought-out design for an application program. Since there are generally more students who can program the computer than who have good ideas about what to get the computer to do, the emphasis in the course is upon the development of a good idea, unencumbered by the limitations of having to actually implement the idea. This emphasis also helps to ensure that students with weak programming skills are not handicapped. In fact, if anything, the handicap seems to be on the more able programmers. These students have a tendency to constrain or limit their design ideas to those they think they could actually program by themselves.

The student software architect develops the specifications for how his or her program will interact with a user, ignoring the programming required to get the program to engage in that interaction. For example, in recent terms students have suggested and worked out preliminary design architectures for the following: a Filevision-like application for use in keeping track of roommate preferences and making dormitory room assignments (Filevision is a database program for the Macintosh that utilizes icons and their locations to represent objects and relations in the data base); two different versions of a course scheduler that enables the user of the program to optimize a class schedule both within a given term and over an academic year; a menu/meal planner to assist in scheduling and sequencing preparation of a complete set of meals over the course of a week; and a redesign of some of the courseware currently used in Drexel's general psychology course.

Impact of Application Programs

It is in the development of the architectural specifications for student course projects that microcomputer application programs become particularly important, both as examples and as tools. One course exercise is to critique an application created at Drexel University (typically the microcomputer project disk). This exercise provides a focus for the discussion of design issues and psychological principles in the early part of the course, and helps to create a starting point, later in the course, for discussion of the processes of evaluation and redesign of software.

Using the Drexel Disk as the object of this critique has continued the involvement of students from the software psychology class in the interactive disk concept originally worked out by earlier students. It has also contributed to evaluation and refinement of the program and the course. For example, the second version of the Drexel Disk was done after the first version had been given a thorough review by students in the software psychology class. (A particularly instructive part of this experience for me was that this review and evaluation were done without the students' realizing that I had been involved with the development of the application they were critiquing.) In addition, some students have chosen to focus their design efforts on a redesigning of the Drexel Disk, or on the development of ideas for significant new features to be added to future versions.

Other application programs, such as word processors and graphics programs, become particularly useful as design tools. Obviously, the word processor can be used in writing a report for the course. Of more interest is the role of graphics programs. One important feature of the software architect's role is the responsibility for development of the initial draft versions of project planning documents to be provided to the builder. These project planning documents should contain a description of the complete architecture of the software, focusing primarily on the nature and sequence of the interaction between user and computer, and a general statement of the objectives of the programming work to be done. A useful tool in this context is the development and revision of "storyboards." Storyboarding is a technique borrowed from filmmakers. The software architect uses storyboarding to create a visual representation of the sequence and nature of user-computer interactions by providing "snapshots" of crucial screen displays or choice points in the interaction.

Students in the software psychology course are encouraged to take advantage of storyboarding by allowing them to substitute pictures of screen displays for text descriptions of user-program interaction. Given this encouragement, many students incorporate the relatively sophisticated graphics made possible by such Apple Macintosh programs as MacPaint and MacDraw. In fact, some course projects have consisted almost entirely of a sequence of screen displays threaded together by a brief narrative.

The Interaction Effect

This environment also provides an excellent context for exploration of the meaning and application of psychological principles. For example, one discussion in the most recent offering of the software psychology course focused on the degree of breadth and depth that should be built into menus when they are used to control application programs; that is, should a menu tree structure have several short branches, or a few long branches? Some of the students had had a programming course in which they were told that short-term-memory capacity was limited to seven items, so they should not use menus with more than seven alternatives. This claim led to a class discussion of the implications of a short-term-memory capacity limitation for the problem of choosing between menu breadth and depth when designing a menu structure.

As the discussion developed, one student observed that the argument for limiting the number of menu alternatives to seven items because of a limitation on short-term-memory capacity did not seem quite right. His reasoning was that the menu alternatives are typically visible on the screen display and are not actually held in the user's short-term memory. Another student, picking up on the idea that the menu alternatives are visible to the user, suggested that each menu alternative, being visible, is actually a cue representing a "chunk" of information, and that the chunk contains either a submenu with additional commands or a course of action the computer takes in response to the menu command. She also noted that any user memory problems would appear to be related more to the organizational basis for chunking, to the effectiveness of the cue, and to the size or complexity of the chunk than to the number of visible menu items. It was not long before another student suggested that, since chunks appear to be variable in size, the latter observation seemed to imply that a fundamental issue in developing a menu structure for a computer application program is really the finding of a simple, coherent, easily remembered organizational basis for chunking those portions of the command menu structure that are not visible on the screen.

CONCLUDING REMARKS

One consequence of a computer-rich environment is that the computer and its power are no longer a scarce resource to be jealously hoarded and shared out on a controlled-access basis. Rather, a student or instructor who chooses to spend a large amount of time exploring the complexities or applications of a piece of software is able to do so at his or her own convenience, without depriving anyone else of access. This freedom of access also means that the instructor can develop new ways of using the computer and its application programs as a tool for working with the raw material out of which knowledge is created.

In a very real sense, a new opportunity and a new challenge are created by the computer-rich environment. The new opportunity lies in introducing a computer-based component into the entire psychology curriculum, tying together the curriculum in new ways that introduce a new kind and level of experience for faculty and students. The new challenge lies in reexamining the contents of the curriculum, looking for things worth doing on, with, or by the computer that enable both faculty and students to see and use computer technology as a tool in learning and mastering their chosen field of endeavor.

REFERENCES

Brooks, F. P., Jr. (1975). *The mythical man-month*. Reading, MA: Addison-Wesley.

Castellan, N. J., Jr. (1983). Strategies for instructional computing. *Behavior Research Methods & Instrumentation*, **15**, 270-279.

Chute, D. L. (1986). MacLaboratory for psychology: General experimental psychology with Apple's Macintosh. *Behavior Research Methods, Instruments, & Computers*, **18**, 205-209.

Collyer, C. E. (1984). Using computers in the teaching of psychology: Five things that seem to work. *Teaching of Psychology*, **11**, 206-209.

Hewett, T. T. (1985). Teaching students to model neural circuits and neural networks using an electronic spreadsheet simulator. *Behavior Research Methods, Instruments, & Computers*, **17**, 339-344.

Hewett, T. T. (1986a). The electronic spreadsheet as a tool in modeling and simulating neural networks. *Proceedings of the Conference on Modeling and Simulation on Microcomputers* (pp. 165-169). San Diego: Society for Computer Simulation.

Hewett, T. T. (1986b). Using an electronic spreadsheet simulator to teach neural modeling of visual phenomena. *Collegiate Microcomputer*, **4**, 141-151.

Hewett, T. T. (1986c) When every student has a computer: A new perspective on courseware and its development. *Behavior Research Methods, Instruments, & Computers*, **18**, 188-195.

Hewett, T. T., & Perkey, D. J. (1984). The mythical "mountain" of software. *Collegiate Microcomputer*, **2**, 207-210.

Hewett, T. T., Perkey, M. N., & Wozny, L. (1986). *The Drexel Disk© 2.5* [Computer program]. Philadelphia, PA: Drexel University.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books.

Perkey, M. N. (1986). The effect of a machine-rich environment on courseware development: The process and the product. *Behavior Research Methods, Instruments, & Computers*, **18**, 196-209.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems*. Cambridge, MA: Winthrop.

Taylor, R. P. (1980). *The computer in the school: Tutor, tool, tutee*. New York: Teacher's College Press, Columbia University.

Wozny, L. A. (1986, May). *Painless information retrieval: The effect of good user interface on design*. Presented at the 15th Mid-Year Meetings of the American Society for Information Science, Portland, OR.