A Survey Course in Computer Science Using HyperCard

Richard W. Decker and Stuart H. Hirshfield

Department of Mathematics and Computer Science
Hamilton College
Clinton, NY 13323

Abstract

This paper describes a survey course in computer science and the materials we have developed to support it. The course is distinctive in at least three ways. First, it presents a comprehensive, disciplinary survey of the field. Second, it emphasizes the principles upon which the discipline is based, stressing liberal education as opposed to technical training. Finally, it reflects our belief that one learns best by doing, as well as thinking, and so includes customized software in the form of HyperCard stacks to support an integrated, directed laboratory component. Our experience over the past year indicates that the audience appreciates this survey approach, that students come away with both a sense of accomplishment and a realistic feeling for the breadth and substance of the discipline, and that HyperCard is a superb medium for illustrating, demonstrating, and making accessible a wide range of computer science topics.

1. Problems With the Survey Course

The problem with a survey course in computer science is twofold. First, there is no all-encompassing metaphor for computer science. Even the accepted definitions of the discipline admit to a variety of legitimate perspectives, and point to a multi-faceted, diverse field of study. Indeed, it appears that most of us are likely in the same position regarding computer science as Justice Stewart was regarding pornography: we may not be able to define it, but we know it when we see it [4].

Not surprisingly, the second problem has been—and continues to be—that there is no agreement about which perspective is appropriate for the survey course. In physics, chemistry, geology, and mathematics, in contrast, there is at least a partial consensus among professionals about the curriculum of the survey course, based upon decades or centuries of experience.

The first exposure to computer science, on the other hand, appears in many guises at different institutions. In the relatively brief history of computer science education, three models have emerged as the most prevalent forms of introductory course. We characterize these models as follows, noting that we paint these approaches with rather a broad brush. If we err on the side of caricature, we apologize in advance—we do so only for emphasis.

• *All About Pascal [except pointers]*. Probably the most common model, this course often serves the dual purpose of doing something computer-related for the general audience, as well as providing an entry point for the computer science major. Programming and algorithmic problem-solving are important subjects, and should indeed be encountered very early in the career of the computer science major. As a course for a general audience, however, this approach has several failings. First, it does nothing to dispel the notion that "Computer science = Programming." Typically, students are not exposed to the important notions of computer science, except perhaps osmotically. This failing is even more poignant for potential computer science majors, who come out of their first course having almost no idea of what lies ahead, and in near complete ignorance of the fact that what lies ahead has very little to do with what they have just seen. A second problem is that this course is typically taught in Pascal or C (or perhaps Modula-2 or even Ada). For many students at the introductory level, the syntax and semantics of a high-level language stand squarely in the way of the learning process: even students with very good quantitative skills often expend more energy learning where semicolons belong than they do mastering the concepts of designing algorithms. Those with lesser quantitative skills often find programming wholly inaccessible and come away firmly (and perhaps wrongly) convinced that computing is not for them.

• *The EDP/MIS Approach, or How to Use_____ [fill in your favorite set of microcomputer applications]*. In an attempt to provide something of practical value

to the students, such courses typically cover the use of a word processor, a spreadsheet, a file manager, a database program, and perhaps a graphics program and communications software. From a purely practical point of view, if for no other reason, this approach is problematic. The most significant difficulty with this approach, though, is its concentration on *training* students on implementation-specific details at the expense of *educating* them in the principles which underlie text processing and database theory, for instance. Not only is it more appropriate that this material be offered as one or more short, non-credit courses within the Computer Center, but it is also very likely that any expertise in a particular spreadsheet will be antiquated by the time the student graduates.

A slightly more substantial variation on this theme emphasizes computer management skills. This course typically includes units on the stages of system development, a comparison of centralized versus distributed data processing and how to decide which approach is best for a business, considerations when entering into a contract with a vendor, and descriptions of career opportunities.

This approach can indeed be justified in some programs which are more vocational- than liberal arts-oriented. It is, nonetheless, not what we have in mind when we consider a survey course in computer science. The emphasis in this course often leans more towards using computers and managing their use than to understanding basic principles and areas of concern in the discipline.

• *The Soft Side of Computers.* Stemming from a social science perspective, this course explores the impact of computers on the modern and future worlds from social, political, economic, and historical points of view. Done well, this is a valuable approach, in spite of the fact that computer science itself is usually given short shrift in an attempt to look at the effects of computer use. Unfortunately, this multidisciplinary point of view is very difficult to do well, thus running the risk of covering a little bit of everything, and none of it well. The result can be that the course material degenerates into a collection of anecdotes culled from the popular press, with no attempt at a serious disciplinary point of view from any of the disciplines involved, least of all computer science. Such courses often expend a great deal of energy apologizing for the technical nature of the field and almost invariably adopt a condescending attitude towards the students. While we appreciate the potential value of this model, we also believe that understanding the social impact of technology requires an understanding of the technology itself.

2. One Solution

Our approach to the survey course in computer science is based on three fundamental ideas. First, we approach this course from a liberal arts perspective; second, we want to maintain a serious, disciplinary point of view, and, finally, we designed the course so that laboratory exercises are an integral part of the entire experience. We designed the text and software with these three points in mind, and believe that the resulting combination represents an entirely appropriate solution to the problems of exposing a lay audience to computer science.

When we talk about a liberal arts perspective, we mean that we see our mission as educators as one of preparing our students to become educated and productive citizens, capable of adapting to change and making informed decisions about the problems of the world in which they will live. For the purpose of our survey course, this implies an understanding that in an increasingly technological society—particularly one in which the computer and its programs will play a potentially large role—citizens will need to be informed of the nature of the technology and the points of view of its practitioners. Technology does not exist in a social vacuum. Although the study of algorithms can demonstrate what computers can and cannot do efficiently, and the study of computability theory can show what computers can and cannot do at all, we should not neglect the importance of questions about what computers *should* and *should not* be used to do.

Further evidence of the liberal arts perspective is the emphasis throughout the text on principles as opposed to technical details. This is not to say that details (e.g., of HyperCard, HyperTalk, etc.) are neglected. Rather, they are presented in the context of more general—and important—topics (user interface, programming), and only to the extent necessary to explain examples and the sample stacks. Students are encouraged to explore supplemental documentation and descriptive materials as part of their laboratory exercises.

The disciplinary nature of the course is also evident throughout. Naturally, a course such as this includes an introduction to programming, algorithmic problem-solving, and design paradigms. It also introduces many of the "classic" data structures and algorithms, program translation, machine organization, operating systems, artificial intelligence, and computability theory—in short, it provides a glimpse of most of the major topics in computer science.

The course is lab-based because we firmly believe that computer science, like the other sciences, is not a spectator sport—it is a contact sport. Each of the nine units of the course has an associated lab component in the form of one or more HyperCard stacks. These labs are an indispensable part of the course, serving to reinforce the text, where reading takes place, by providing an environment where *doing* takes place. Another rea-

son why the course is lab-based is a very simple one: it works. The introductory computer science courses at Hamilton, in one form or another, have been lab-based for eight years, and our experience has convinced us that this is method is not only applicable for skill acquisition such as learning a language, but also for reinforcing more abstract concepts, as well.

3. The Analytical Engine

The two most common responses we have had from people to whom we have described our course are "Why did you decide to use HyperCard?" and "What is your syllabus?" The syllabus is easy enough to describe; our reasons for using HyperCard take a bit more explanation.

The Answer is HyperCard, the Question is ... ?

When Apple Computer introduced HyperCard in 1987, the new product was greeted with enthusiasm, albeit mixed with a touch of bewilderment. Apple classified HyperCard as "system software," but that description satisfied almost none of the early reviewers. It wasn't just a paint program, it wasn't just a database management program wrapped in a slick graphics package, it wasn't just a programming environment—in fact, HyperCard seemed to be a solution in search of a problem.

We feel that one problem for which HyperCard is ideally suited is courseware development. Using HyperCard stacks for the lab modules in our survey course has several advantages. The most important advantage is that the entire package is easily accessible to novices. A common complaint among students learning a high level language for the first time is that they fail to understand why they should spend hours writing and debugging a program to do something—like sorting a list of numbers—which they could do by hand in a few minutes. These students are rarely appeased by the instructor's response that in designing, testing, and debugging their programs they are learning valuable cognitive skills and gaining experience in algorithmic problem-solving. Beginning programmers want results, and we feel that at the introductory level this is an entirely appropriate point of view. In authoring a HyperCard stack, the students find that they can do in minutes something that their contemporaries in an introductory Pascal course could not do in days of work. For our purposes, this means that instead of simply teaching our students how to use a spreadsheet or database program, they can design or modify one of their own, without ever having to consider the intricacies of the Macintosh's file protocols or the List Manager ROM routines. The fact that HyperCard handles many of the programming details behind the scenes means that the students' interest stays at a high level, and also allows them to concentrate on problem-solving, rather than having to be overly concerned with programming details.

It also means that the topic of programming can be properly motivated in terms of a student's experience using and designing stacks. Having become HyperCard "authors," concentrating as they do on issues of user interface and system design, the students find that all of the code which was generated for them is easily accessible for inspection and modification. The HyperTalk programming language, while being considerably more verbose than Pascal or C, has the advantage that it is much more natural for the beginning user. Because the HyperTalk interpreter often admits several alternative ways to express a single statement, we find that our students have less anxiety writing programs in HyperTalk than do students using a more compact, but less "forgiving" language. In addition, all of the HyperTalk scripts associated with a stack can be inspected in source code form, in much the same way that the Smalltalk system browser makes the source code for all methods public. We have found that students learning any language can often find their way out of a problem more quickly by seeing examples of code—particularly from programs they are familiar with—than they can by listening to descriptions of principles.

HyperTalk has other advantages which, frankly, didn't surface until we had had some experience using it as the programming vehicle for this course. By exposing our students to object-oriented programming and hypermedia at this level, we are introducing, more or less painlessly, topics which are at the cutting edge of computer science research. HyperTalk is not truly object-oriented in the formal sense of the term, but even in this simple form we have encapsulation of data and methods and a rudimentary form of message-passing, two of the fundamental concepts of object-oriented programming.

What's it All About?

The course is divided into nine units, each with its own metaphor. We begin by going down, descending from general to specific, or, if you will, from abstract to concrete. Having reached the lowest level, that of hardware and architecture, we progress upwards in generality, from the theory of abstract machines, through the prospects of intelligent machines, to speculations about the future. In the following paragraphs the text and disk components of each module are described, in order and in more detail.

a. History. In this introductory module, we place the computer in an historical context, beginning with Babbage. The Analytical Engine serves as the metaphor for this module, embodying as it does the notion of an autonomous information processing machine. We place the Analytical Engine in the context of the "skilled ma-

chines" of the time and discuss the reactions to the introduction of such machines. With a brief nod to Herman Hollerith, we skip ahead a century and consider the early computers of Atanasoff and Zuse, leading up to wartime and its impetus on the development of electromechanical and electronic machines. The invention of the transistor leads to a discussion of generations of hardware, which we carry to the present.

The disks which form the lab portion of the course contain a custom Home Card which leads the students to the "Starter Stack," designed to familiarize the students with the basic navigation on the Macintosh and HyperCard. Figure 1 shows a typical card in the Starter Stack.

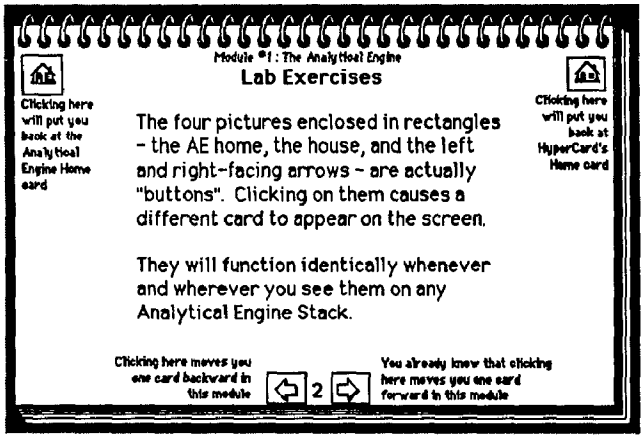

Figure 1. Starter Stack

b. Applications and Implications. While the first module concentrated on hardware, this module treats programs from a "black box" point of view, by introducing calculators, word processors, and spreadsheets and explaining them in some detail. Our point of view follows the first module in a natural way, asking "Now that we have all this power, what do we do with it?" Using the calculator as our metaphor is particularly appropriate here: first, it allows us to discuss the explosive and unforseen growth of microelectronic technology, second, since the program which runs an electronic calculator is built in at the factory, a calculator is truly a black box, and, finally, the calculator allows us to discuss concerns about deskilling, productivity, and information storage in an age of technology. We consider the familiar applications—and those of a more specialized nature in science, business, and the professions—as microworlds which often begin as models of familiar applications (early text processors as models of a typewriter, for instance), but which often evolve functions far beyond those of the original application (a modern word processor allows font changing, spelling checking and automatic word counts, not to mention the ability to embed graphics in the body of the text).

The lab disk contains a calculator, a word processor, and a spreadsheet for the students to explore. Later, the students can return to these applications and inspect and modify their code, but for the time being we treat these without concern for how they accomplish their functions.

c. Designing for Use. Continuing our progress towards more detail, this unit covers system design by concentrating on the design of a user interface. Our metaphor here is the hippogryph, a creature made up of parts of other animals. The black box has become a gray box, since we are building a system by combining parts which already have a complicated functionality. At this level, we don't concern ourselves with how the parts work; we concern ourselves only with putting the parts together into a smoothly functioning whole. As the level of detail becomes finer, the user level which determines the students' use of HyperCard also becomes more powerful. Students are introduced to the Authoring level, allowing them to design stacks of their own, and are exposed to the anatomy of a stack as a collection of backgrounds, cards, fields, and buttons arranged in a hierarchical fashion.

In the lab, the students practice what they have read, using the painting tools and other authoring features to create a stack of their own. They are given a simple database, the "Little Mac Book," which they modify by adding new cards and fields, and changing the look using the painting tools.

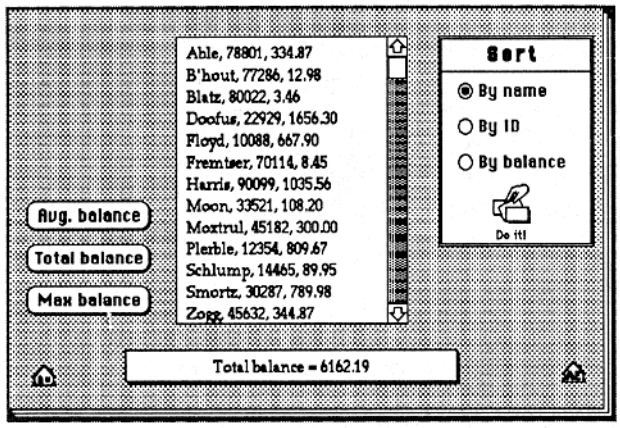

Figure 2. Sorting Card

d. Programming. We now make the gray box into a clear box. We adopt a culinary metaphor here: using an application is similar to being a guest at a dinner party, where the sequence of courses and their contents had been chosen beforehand, while designing a system in the previous unit is equivalent to being the host at the party, responsible for choosing the courses. In this unit we take the students into the kitchen, as it were, showing how to produce the courses by suitably combining the ingredients. The students continue here their exploration of the levels of HyperCard, moving to the

Scripting level which allows them to access and modify the programs of a stack. In the text part of this module we explore the information and control structures by designing a selection sort script for a button on a card. Figure 2 illustrates the sorting card, which the students are encouraged to explore. At this point, we also cover the object hierarchy of HyperCard and show how messages are passed between objects and used by message handlers. We conclude with a look at software design paradigms and the software life cycle.

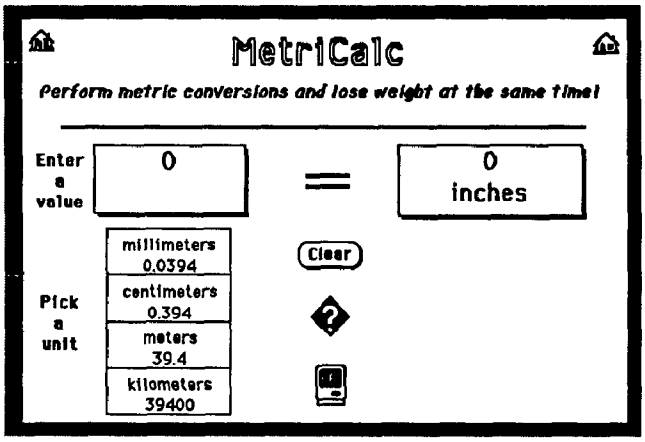

Figure 3. Unit Conversion Card

Figure 3 illustrates another of the sample stacks used in the lab portion of this module, a simple unit conversion program. The lab consists of a large number of directed exercises in modifying and extending this stack.

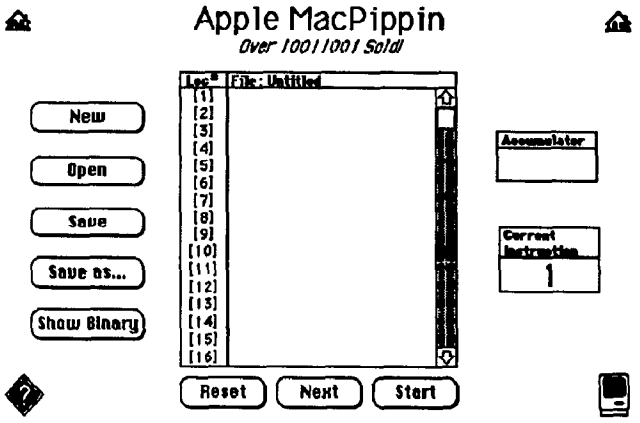

Figure 4. Assembler

e. Program Translation. In this module we investigate how a computer can be made to execute programs in high-level languages such as Pascal, LISP and HyperTalk. Using the Rosetta Stone as our metaphor, we introduce binary representation of information, and discuss the machine language of a hypothetical computer and an assembler language for that machine. We provide a survey of several language paradigms and conclude with a brief introduction to the problems of program translation.

The lab part of this module includes two stacks. In the first, the students explore the conversion from character strings to strings of ASCII codes, and then from these codes to their binary representations. Figure 4 illustrates the second stack, an assembler/executor for the language introduced in the text. This stack allows the students to enter an assembler program from a file, see it translated into machine code, and step through the execution of the program.

f. Hardware. This module is a natural next step after the previous one, since the students have already seen hints that a computer represents a machine language program as a collection of binary data. In this module, we show how machine language statements are executed. In keeping with the pervasive notion of levels of abstraction, we use a simple light switch as a metaphor and see how gates may be made by combining switches, how gates combine to make circuits, and how circuits are combined in the architecture level of a computer. As a further link to the previous module, the text part includes a construction of a complete, if simple, computer which runs the assembler language programs introduced in the previous module.

The lab stack for this module is a simulated breadboard which can be used to build and test combinatorial circuits made by connecting AND, OR, NOT, and NAND gates.

g. Theory of Computation. This material is conspicuous by its absence in most survey courses. We feel, though, that it is important for students both to know there is a theory which lies beneath the practical details of computation, and to realize that, despite being conceived half a century ago, the theory prescribes hard and fast limitations on today's computers. It is also valuable for them to see that there are problems which sound like reasonable candidates for automation but which cannot be solved by any (Turing machine) program. The Turing machine serves as the metaphor for this unit, which provides us with a link back to the earlier unit on history as well as a forward one to the next module on artificial intelligence. We take two points of view in this module, viewing programs as partial functions on the set of all binary strings, and as programs for Turing machines. The usual diagonalization argument is used to show the existence of partial functions on binary strings which cannot be realized by TM programs, and the Halting Problem is used as an example of a problem which sounds as if it should be amenable to TM solution, but is not.

Figure 5 illustrates the lab stack for this module—a Turing machine simulator. The lab also includes files of TM programs which have been written for the simulator, demonstrating that, for example, a TM can add

and move information from one location on its tape to another.

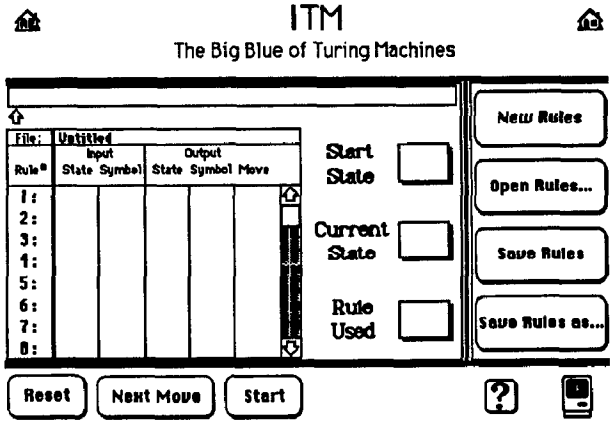

Figure 5. Turing Machine Simulator

h. Artificial Intelligence. Continuing the upward turn in focus which began with the previous module, this module expands the level of abstraction to representing intelligent behavior in a program. The metaphor for this unit is the HAL 9000 computer, described in Arthur C. Clarke's *2001*. We make the point that the things humans do most easily, like natural language recognition, are the most difficult to embody in a program, and the things we have to work hardest at, like playing checkers, are generally easier to mechanize. HAL is used throughout as an exemplar of language and visual processing, knowledge representation,and learning. We provide rough comparisons of the quantifiable differences between brains and machines, and we take pains to emphasize that HAL is well beyond the present state of the art in AI research.

The lab stack for this module is a simple poetry generator with verse patterns and vocabulary which the students can customize. While the lab uses this stack, as usual, to provide insight into the programming process, we also point out that the stack achieves its behavior using a simple rule-based scheme which is clearly far simpler than the human creative process it simulates.

i. Computers and Society. The metaphor we chose to illuminate the last module is the Sorcerer's Apprentice, and we remind the reader that although that story has a happy ending, we have no guarantee that the Sorcerer will arrive at the last minute to rescue us from the folly of our poor choices. In an attempt to provide a balanced assessment of the social consequences of the computerization of society, we first describe the limitations on our predictive abilities. Then we identify the major trends in computer use—increased power, increased reliance, access to electronic information, centralization, emergent effects—and try to gauge their impacts.

The last lab module contains a simple ELIZA-like program, illustrating the perils Weizenbaum pointed out of anthropomorphizing a program, and a sinister variant which not only behaves as if it were infected by a virus, but also includes code to monitor the supposedly private session between the user and the machine.

4. Details and Context

Our service course has used versions of this package (the text and two lab disks, to be published by Wadsworth in January, 1990) for the past year. Students with no prior Macintosh experience attend an extra lab session during the first week of class to take a Mac Guided Tour, after which they are ready to begin Lab 1. About one-third of our class time is devoted to supervised lab, with the remaining time spent in lecture. Depending upon the module, lectures can focus on (1) topical material, i.e., discussions of topics presented in the text (there is more than enough to support an entirely lecture-based course), (2) demonstrations of sample stacks or other HyperCard facilities, (3) lab-related discussions of particular exercises, or (4) supplemental material—for example, a review of Macintosh applications, viewing related films and tapes (*2001: A Space Odyssey*, John Sculley's "Knowledge Navigator"), or discussing SDI.

Depending on the module, homework assignments can be based on text-related questions, extensions of lab exercises (e.g., use the sample stacks to build a circuit or to write a simple assembly language program), supplemental machine-based exercises (e.g., design a restaurant guide for your home town, write a program to record and balance your check book), or any combination thereof. The class has been taught in small sections and as a large lecture with small labs, and is equally amenable to both formats.

Where does this course fit in the computer science curriculum? A traditional introductory survey course not only serves as a terminal course for those seeking a view of the range of topics in its discipline, but also can act as a proselytizing vehicle, attracting students to the discipline who might not otherwise consider a major or minor in the field. It could also serve well as the first course in a computer science major, either before the standard CS I course, or in place of it. A student who completed this course would certainly be better prepared for CS I than his or her peers, and placing this course before the usual first computer science course would relieve that course from the responsibility of including a large amount of introduction to the topics in the discipline. We feel that the transfer from HyperTalk to Pascal or C does not involve a major paradigm shift—if anything, the object-orientation of HyperTalk points students in the right direction to writing modular programs in other languages. The transition might be made even easier if the CS I course used an object-oriented language such as Object Pascal or C++, for instance.

It is also worth mentioning that each lab module, beginning with Module 4, includes programming exercises related to the corresponding sample stack as well as suggested programming projects. While our present course does not emphasize them, doing so renders the course much more programming-intensive. By the end of the text, students have seen in HyperTalk the topical equivalent of most CS I programming courses (including basic control and data structures, subprograms, parameters, variable scope, predefined functions, etc.) and more (the object hierarchy, self-modifying code, plus all of the survey material).

Because the text and the labs form an indivisible whole, this course obviously requires a computer lab. A number of sources recently [1, 2, 3] have pointed out that computer science, like the other scientific disciplines, is a laboratory science, and we heartily concur with this approach. However, HyperCard is only available for the Macintosh, so this approach inevitably weds one to one machine, at least for the present. There are, however, programs similar to HyperCard for other computers, and we plan to port the course to a different environment shortly.

Aside, though, from having enough Macintoshes to support student lab work, no other facilities are required. It is, to be sure, extremely helpful in explaining HyperCard and demonstrating lab stacks to have a classroom equipped with projection equipment. Otherwise, the package is self-contained. The disks include an appropriate System Folder as well as a copy of HyperCard, boot automatically to a customized Home Card and, when ejected, shut the system down.

5. Evaluation and Summary

Our experience with the package and the course to date indicates, first and foremost, that students appreciate the survey approach. Our service enrollments had dropped dramatically in recent years (our previous "service" course was Pascal programming intensive) and are now on the way up. Students are both relieved that the course deemphasizes programming and are interested to find out that there is more to computer science. Partly because the notion of a lab course is still intimidating to many of them, they feel that the course is demanding. On the other hand, they almost invariably admit to a sense of accomplishment. We attribute this both to the interesting and realistic nature of the problems they are asked to solve, and to the fact that students leave the course completely comfortable with HyperCard and, more importantly, with the Macintosh. They find themselves well prepared to go off and use word processors, spreadsheets and databases with little problem. In short, the package seems to be easy to learn from.

We also think that this package is easy to teach from. The lab approach is not only sound pedagogy, but it renders much of the course nearly self-teaching. This is particularly true if class time is explicitly devoted to lab work. Also, the lab experience, as mentioned, provides additional material for lectures.

Not that it's needed, mind you. Since we decided from the start to commit errors of commission, rather than omission, there is more than enough material for a semester's work. We feel that this encourages flexibility on the part of the instructor, who can choose from a range of topics in designing his or her own version of the course. Indeed, our course does not cover the entire text. Each time it has been taught, a slightly different syllabus has evolved which reflects the interests of current students. Because our approach is somewhat novel, we have prepared detailed instructor's supplements, including lecture notes, transparency masters, answers to exercises, and HyperCard troubleshooting hints.

In closing, we think that this course successfully solves many of the problems associated with a survey course in computer science: It provides a comprehensive overview of the field without pulling any punches and without apology. It is solidly disciplinary in its approach, and emphasizes principles and education at the expense of technical details and training. Finally, it exploits HyperCard as an integrated, expressive and stimulating laboratory environment, particularly for the intended audience.

References

[1] Barker, K., Soldan, D. L., and Stokes, G. E. Laboratory experiences in computer science and engineering. *Computer Science Education* 1(1). 1–10.

[2] Drysdale, R. L. S., Korth, H. F., and Tucker, A. B. Computer science in liberal arts colleges. *Computer Science Education* 1(1). 11–35.

[3] Gibbs, N., and Tucker, A. (Eds.) A Model Curriculum for a liberal arts degree in computer science. *Communications of the ACM* 29(3) 202–210.

[4] *Jacobellis v. Ohio* 378 U. S. 184, 197(1964) (Stewart, J., concurring).