Retraining: Is It The Answer To The Computer Faculty Shortage?

Dr. William Mitchell
University of Evansville
Evansville, IN 47702

Abstract

This paper reports on the experiences acquired in initiating a summer retraining program to prepare college faculty to teach undergraduate computing. The distinction between formal and informal retraining, the benefits of formal retraining, and the justification for credentializing such programs with a masters degree are also discussed.

Introduction

During the mid-80's, and perhaps longer, there will be a critical shortage of academically trained faculty in the computing sciences. The "minimum" academic credential, a master's degree, is already becoming scarce, and increasing numbers of two year and small four year colleges are employing bachelor-level faculty "with experience." The production of the Ph.D. programs in computing is actually declining. But even if that trend is reversed, the demand for people with that credential is so great as to price them out of the reach of smaller colleges and regional universities. One of several ways to respond to this faculty shortage is retraining.

By retraining, we refer to the process by which a faculty member, who is credentialed in some discipline other than computing, acquires the necessary knowledge and skills to instruct courses in the computing curriculum. This process has been going on for some time, and it appears now to be accelerating. Most of the retraining is informal and ad hoc [7].

In some small colleges a faculty member who is pursuing a personal interest in computing, be his area Mathematics, Physics, or English, is given the assignment of teaching computing courses (indeed, of developing a computing curriculum). In larger institutions where computing offerings already exist, new instructors are trained by auditing courses taught by their colleagues. In exceptional cases, these (generally undergraduate) courses are taken for credit. In only a very few cases, the retraining consists of a formal educational program leading to a graduate degree.

We have been aware of the developing competence of colleagues in other departments with regard to programming ability, and it therefore seems quite reasonable to take advantage of this skill and spread the burden of introductory programming instruction which continues to grow. It is therefore not surprising that accounts of the systematic use of such faculty in a variety of colleges should have recently appeared in the computer education literature [2,4]. But as this practice spreads, indeed, as it is recommended, it poses some serious questions about academic quality. This paper will address these questions as it focuses on the distinctions between formal and informal procedures for retraining faculty.

Before we proceed it is appropriate to make mention of a second source of computing faculty: the practitioner. Several voices from the business schools have recently bemoaned the tyranny of the Ph.D. credential which impedes the university from making more systematic use of job-trained professionals [1,5]. These practitioners have frequently been utilized as adjuncts, but now it is suggested that they would make better than adequate professors. Specific mention has been made of both "loaners" from industry and of retirees. As of yet there does not seem to be near as many of these professionals as we have retrainees, but this may be merely because no real effort

has been made to recruit them.

Undoubtly there will be many adjuncts who have already proved their competence to function in the classroom. However, it appears that evaluating an adjunct's ability to perform as a full-time instructor is much more difficult than evaluating the potential of a retrained faculty member. Adjuncts are seldom given very difficult classes to teach, and they are seldom evaluated very carefully. They are chosen for their narrow expertise rather than for their broad understanding of the disciplines. Hiring a professional who has neither academic background nor adjunct teaching experience is indeed risky, even assuming the best environment. To expect that very many computing professionals will have a teaching gift, and will possess reflective and analytic dispositions, is to understand little of the environment out of which they come. James Martin is an oft mentioned exception, but the very fact that he alone is repeatedly identified emphasizes that few other practitioners can emulate him.

A Formal Retraining Program

The author began a masters program in Computing Science Education in the summer of 1982 after two years of experience with a noncredit institute in data processing offered to small college faculty [9]. Both programs are a response to the need for greater computing experience in the small colleges, but the masters program is motivated by the perceived inadequacy of the informal retraining strategies available in the small college. The demand for computing curricula is just as great in these institutions as it is in the universities, but the small college faculty member can neither afford a leave of absence nor sit in on a colleague's course. Therefore a summer graduate program offers the only realistic way that such a faculty member can be retrained (Codespoti and Bays [3] describe the first program of this type known to the author, but it did not automatically culminate in a graduate degree).

The first class consisted of 18 faculty representing 12 different colleges. The program required previous programming experience, a masters degree in some discipline, and access to a college classroom during the academic year between the two summers in residence. The participants included seven mathematicians, two chemists, a historian, a home economist, a social worker, a minister, a musician, a specialist in education, a librarian, a physical education teacher, and a data processing manager who did not have a masters degree or any teaching experience. All but two had had formal coursework in programming, and six had taught programming courses. Four paid their own way to the program while the rest were sponsored by their respective colleges. College teaching experience ranged from none to over 20 years, ages ranged from 23 to over 50, and seven of the participants were women.

During the first summer all students were enrolled in five courses, one running the entire ten-week session, while the remaining four were five weeks each and taken two at a time. Therefore each student was in three classes simultaneously. During the first five weeks these classes were Systems Analysis, Computer Hardware, and Data Structures. During the second five weeks the Data Structures course continued and the new courses were Systems Software and Undergraduate Computing Curricula. Lectures comprised four and one-half hours each day, four days a week. Two of the participants cut back their load by not taking the last two courses. A description of the program from a participant's view may be found in the Winter 1982-83 issue of INTERFACE, The Computer Education Quarterly.

The specific curriculum objectives of the first summer were to lay a broad foundation for the topics of the undergraduate computing curriculum, and to indoctrinate the students in the mores of the discipline and the current issues being debated among computer educators. Furthermore, each participant was required to plan for the course to be offered in the fall term at his home institution, and to critique the curriculum already planned or in place. Transcending these specific objectives, we sought to build in the participants a sense of belonging to the computer profession and to instill a set of values to be communicated to their students. The latter was done by relating the work being pursued in each class to the possible undergraduate presentation of this material. Many of the exercises used during the summer were exercises from the undergraduate courses which the graduate faculty routinely taught. Thus we sought not only to develop the mastery of technical subject matter, but to convey techniques of teaching this subject matter and standards for its evaluation. We tried to make the students conscious of the intellectual difficulties inherent in the topics and to give them opportunity to share with each other the challenges of learning new vocabulary, new modes of organizing information, and new techniques in problem solving.

The summer was completed with mixed results and only after extraordinary effort in the part of several participants and faculty. Our first surprise was how difficult it is to teach teachers. Our second surprise was how little some of the participants' previous formal course work in computing was worth. The best equipped

to program were those few who had been teaching programming. We expected and observed great difficulty on the part of the social science and humanities faculty in reading the text material and understanding the assignments (several had very limited facility for mathematics). Our solution was to provide a great deal of tutoring. To a much lesser extent we observed some difficulty on the part of the mathematics faculty in relating to the context of the systems analysis course. Everyone displayed high ability in verbalizing concepts and questions in the courses which centered about readings and discussion of paper and pencil exercises, but there was a sharp division along disciplinary lines when it came to turning ideas into working programs.

Programming assignments were implemented in PL/C and PL/I, a deliberate choice because it was a language unfamiliar to all participants, and an appropriate one for the data structures course. Eight programming exercises were assigned in that course and one programming exercise was part of the final two weeks of the systems software course. Of the nine exercises, the most completed was seven and one-half, the least completed was one. As a consequence, these two courses were the only ones in which any students received incompletes. A brief description of the programming assignments is included in the appendix.

Even though student performance in some areas was not up to our desired level, we were impressed by how much the participants learned and by how hard they worked. We recognized from the onset that our standards could well be unrealistic, and that our goal could be unachievable. We constantly reminded ourselves that it wasn't necessary that these faculty learn everything their first summer, but that they be adequately prepared to continue learning as they presented their courses in the academic year. We emphasized to them that the habits of study that we sought to forge in the summer would serve them well during the academic year.

The participants returned to their colleges, and each is now engaged in teaching computing. In the fall term half taught either a literacy course or an introduction to data processing course which had very little programming. Four taught a BASIC programming course as a first course to both majors and nonmajors, and thus had a seconday goal of literacy. The faculty being retrained at UE, (the librarian, the education specialist and the musician) team-taught courses in structured systems analysis, data structures, and PL/C programming, respectively, with the assistance of senior faculty. Two other participants taught FORTRAN and COBOL, courses which they had taught before (a total of five of the participants taught courses which they had taught before). Without exception, all faculty were comfortable with their assignments and were considered to be doing well by their supervisors. In six cases these participants were the principal computer experts on campus. All reported that their summer work had effectively prepared them for their responsibilities.

While this measure of success is gratifying to the participants, especially for the eleven who had never taught a CS course before, it is fair to emphasize that most of the courses they taught were not conceptually difficult, and the students they instructed were not very sophisticated or demanding. Yet even if a course is mostly an elaboration on the vocabulary being presented in the text, and even if the faculty member sticks like glue to the instrutor's guide, it remains true that most of these faculty designed and presented a course which they had never seen before. It is also true that each faculty member understands the function of his course within his curriculum, the objectives of the assignments, and the standards to which he holds his students. Each is able to correlate his text with others (and makes a practice of doing so), and each feels competent to discriminate among the differing perspectives offered by the various text authors.

We will have another summer in which to refine the perceptions and skills of this class. We will look for a greatly increased skill level when they return, especially from those who were weak the first summer (the weak students all audited programming classes the Fall term, and they will all teach introductory programming in the Spring). In the second summer we plan to examine in practical depth the characteristics of programming languages, how they are designed and how they change. Students will also have a chance to elect two computer courses which suit their interests.

Reflections on the Advantages of Formal Retraining

To form a preliminary assessment of the accomplishments of this program the author has reflected on several forms of data, all informally collected. There were the impressions built up over hours of work with these students during the summer, including counseling and advising sessions. There were the discussions in the curriculum course, the individual course designs and the curriculum critiques. There were the formal evaluations at the end of the summer. Finally, there were the visitations to each college during the academic year at which time the author observed the

participating faculty member in the classroom and visited with his dean and department chairman. These impressions, together with the intimate involvment with the four who taught in our department and the experiences we shared in our new instructor seminar, have formed the basis for the following comments.

Could these same outcomes observed above be achieved informally? The five faculty who repeated the courses they had offered previously did not change texts or syllabi, but they found that they understood the courses differently this time through. Without a formal training period, it is unlikely that this new perspective would have been achieved as quickly (if at all). For most of the others, it would have been several years before their own self-study efforts would have brought them to their present level.

The participants mentioned most frequently the unifying nature of the curriculum course in solidifying their summer work. The directed readings and discussions of that class were well out of their reach as individuals because the information is dispersed and none of the colleges had access to the computer education literature. Yet the curriculum course was rooted in the experience and knowledge of the other four courses. We could not have discussed the importance of algorithms, of vocabulary, of notation, of evaluation, of curriculum balance, of professional goals, or of learning problems if we had not experienced each of these concepts through the summer.

The textbooks could have been read, good and bad exercises could have been done, and even insight could have come through these activities when pursued in isolation. But the concept of curriculum is not easily grasped in this way. Similarly, one might sit in several undergraduate classes before one had seen enough to begin to perceive the shape and structure of the curriculum. Disregarding the advantage in time which formal training enjoys (offset by its expense), we argue that the informal methods will almost always be critically incomplete in producing a "big picture." The participants of the formal program have acquired insight which directs their growth and makes them aware of their limits. They know what is important in what they will teach and are not prone to become fixated on details or trivialities. They have gained control of their subject matter, even if they do not yet have mastery, and this gives them the confidence to be the expert within their limited spheres.

Both the formal and informal retraining processes seek to create new "computer people." The informal process, if allowed its own way, would make that selection by rewarding ability and capturing the imagination of those with the persistance and discipline to overcome obstacles of vocabulary and the inflexibility of computer systems. The formal process may deal with those who lack the ability to profit by informal experiences. The formal process may be much more effective in the short run, but can either retraining process be said to be truly effective in the long run? Can one really become a competent instructor of computing if one does not have much talent for it, or if that inborn talent is not cultivated in the normal graduate regimen? And if so, will not the retrained faculty member, if that training is not of the caliber of the recommended MS and Ph.D. in computer science, be at best a temporary stopgap?

We make no pretense that the coursework of our retrained faculty members is comparable to that of an MS in computer science, much less a Ph.D. The required core of our program does compress the majority of concepts treated in a rigorous bachelors program into six courses, and it requires the development of a respectable level of skill in applying this knowledge (for a discussion of the feasiblity of this type of compaction, see Sharma and Behforooz [7]). But remember that we are not preparing researchers or even scholars (although two papers of publishable quality are required in the program). The faculty who are candidates for retraining are virtually all devoted to teaching, and they are seeking to be prepared to teach the courses of the lower division of the undergraduate curriculum. The goal of the retraining program is met if the faculty member is proficient in presenting and demonstrating the computing concepts appropriate to this level. In particular, these faculty will be called upon to convey the principles of software development and to survey the context in which this activity occurs. They will also bear significant responsiblity for providing opportunity for general computer literacy on their campuses.

If retrained faculty progress to the courses of the upper division, or to graduate courses in computing, it will be on the basis of continued personal development after completing their formal studies. We anticipate that those faculty with research training will most likely migrate toward more abstract topics in computing, while those faculty with teaching-oriented training will progress to those topics only if they discover an affinity for the concepts and methods of computer science. Undergraduate computing may be much like cooking, where almost anyone can achieve comptetence in the basic arts and learn to prepare credible meals by recipe. Occassionally, an amateur chef is discovered. But good

recipes and a foundation in the use of cooking utensils and the elementary techniques of food preparation are sufficient background to begin to teach those who have never boiled an egg. Analogous to the memory systems of the machines we use, computing education can effectively utilize a multi-level instructor system, so that the different phases of instruction can utilize diversely prepared faculty. At issue then is the minimum faculty competencies we should expect at any given level of the computing curriculum.

In our opinion, the easiest course to teach is a coding course in which a student is presented the syntax of a programming language. The prerequisite qualifications for the instructor of such a course is familiarity with the language in question. This familiarity might well have been acquired the term before when the instructor audited the course (this is more believable if this is not the first language which the instructor has acquired). The instructor has control over the exercises, so he can usually keep the students in the area of the language he has mastered. With the help of the language manual, he will be able to say what should be written, to recognize invalid logic, and to interpret error messages. It would, of course, be better if the instructor had a rich experience in the language and was able to explain the use and function of every facet--the why as well as the what. The class would then be more interesting, especially for the gifted students. But in this level course it is seldom necessary to stray far from the textbook, so this degree of competence is seldom appreciated.

On the other hand, a course in programming is much more difficult to teach because it seeks to integrate problem solving and language use. The instructor in such a course should not only be familiar with the language but familiar with appropriate problems and the techniques which are emphasized in the language. Now he must know not only something about syntax but something about functionality, elegance and style. Now he must focus not only in the language and its valid statements, but on problems and their most effective solutions. Clearly this will require an understanding of both the practice and acquisition of problem solving skills and software development skills in addition to familiarity with syntax. In the absense of such an understanding the programming course becomes a coding course, a common tragedy of the past, and one which will be repeated untold more times into the future.

As we examine the other courses in the computing curriculum we will recognize here and there the necessity to have had certain experiences, the need to have grappled with some deep concepts, the need for perspective and integration as a prerequisite for an effective presentation of concepts so as to neither overwhelm the student nor preclude elaboration. We will also find many topics which are straight forward, relatively uncomplex, and which are necessary to experience and accumulate in order to base a leap to the next level of understanding.

Within the limited time of a formal program we can judiciously choose to provide an array of experiences which will give the retrained faculty member significant intellectual momentum. We believe that the most significant concepts involved in pursuing computer applications are accessible to most graduate-trained faculty if they have the desire to acquire them. (The reader may wish to investigate the very similar core courses which have been selected in the different retraining programs to date. In addition to those referenced in this paper, consider the nascent programs at Memphis State University and Clarkson College of Technology.) We have struggled with faculty who displayed little acumen, but whose perserverance won at last a glimmer of understanding and the promise of proficiency. These faculty have matured in the classroom, and we now have evidence that such faculty can serve effectively in assisting students to acquire a foundation for computer studies. There is no reason why such retrained faculty cannot continue to grow in the understanding of their new role and in the knowledge of their subject matter, adjusting to new texts and to inevitable curriculum revisions. Their contributions are critically needed today and the need will continue throughout the foreseeable future of their colleges. Given an informed perspective and a set of values to communicate, these faculty will continue to serve as planners and interpreters even after more technically proficient staff are acquired. These faculty may be very limited in their range of competence, but what is critical to their usefulness is not their scope but the perspective which is afforded to the students in their classes. These faculty must lead their students to develop accurate understanding and asthetic sensibilities about computing. While these faculty may not themselves be original, they must be educated to recognize and reward originality in their students. Within the context of specific courses, we believe that this can be accomplished.

The easiest group of faculty to retrain will be drawn from the cognate disciplines of mathematics, the physical sciences, economics and business since these are the areas which already support the majority of computer applications. These faculty enjoy better conceptual

groundings for assimilating the available literature. But the real strength of the movement to retrain faculty is that it will include faculty from the nontraditional disciplines as well--and these faculty will bring with them new perspectives in teaching, learning, and using computers. These faculty will enrich the computing education field with their insights, and will extend the acessibility of computing more effectively than has the scientifically-oriented scholar. But because these faculty are more difficult to retrain, we run the risk of sacrificing their potential contribution for the convenience of working with faculty who will progress faster and more smoothly.

The retrained faculty should have earned their status within their institutions in their original disciplines. The acquisition of a second credential cannot diminish that status, but has every likelihood of enhancing it. We therefore suggest that a retraining credential is credible and, after the faculty member has proved himself in his original discipline, it is more appropriate for faculty of certain teaching institutions then a non-teaching-oriented MS in computer science. It is too common for the MS in computer science to permit specialization and to assume that computer skills and intuitions are posessed which undergird the specialized concepts. Even if due consideration is given to the spotty preparation of the retraining MS students, the MS in computer science is not oriented toward preparing teachers, but researchers and practitioners. Programs which capitalize on the background of the faculty member and pursue his retraining goals are going to be more effective in equipping him for his chosen task.

There will continue to be debate over whether retraining should be credentialized, or whether that credential should be a graduate certificate or a graduate degree. We have argued for a distinct degree from the MS in Computer Science. The motivation for awarding a degree is based on two practicalities. The first is that the time, effort and expense involved in formal retraining is comparable to the traditional masters degree, and therefore a degree is the most adequate form of compensation, even if the content differs from the MS in Computer Science.

The second practicality is that the masters level credential is the minimum acceptable level for teaching in an undergraduate major. The uniqueness of retraining is that it is a second masters, and as such is relieved of the necessity of filtering out those students who lack academic dispositions. It fulfills the role of a graduate minor in a doctoral program. There is little to inhibit a Ph.D. from teaching in his minor area.

The choice between a certificate or degree may not be important to those who already have a Ph.D., even if it is not in a traditional cognate area (today, computing is cognate to every discipline), but we predict that the bulk of those seeking formal retraining will not have Ph.D.s. The Ph.D. holder is more likely to retrain informally. The formal retraining programs will be subscribed by the small colleges who still have barely half Ph.D.-level faculty, and these Ph.D.s are least likely to be expendable in their native departments. The additional structure and standards-monitoring which a degree program is likely to enjoy over a certificate program is an edge which we cannot afford to ignore when we anticipate retraining masters-level faculty.

Conclusion

The process of retraining faculty to teach in the undergraduate computing curriculum is already entrenched and is growing. We have sought in this paper to present an understanding and evaluation of this phenomena. Society will certainly strive to increase the numbers of traditionally trained faculty, and in some institutions there will be little desire for the retrained variety. But especially in the smaller colleges, there will be a period in which there will be little other then retrained instructors, followed by an extended period in which both the traditional and the retrained faculty will have to share the burden of delivering computing education. We have presented the accomplishments of a new, formal retraining effort for small college faculty, and have stressed the desirable characteristics which the informal process of retraining, even at its best, is unlikely to achieve. We have argued for the integrity of the retraining degree, and tried to clarify the goals to which it should aspire in distinction to the traditional MS in computing science.

References

1. Athey, Thomas, "Computer Education Challenges of the 1980s," INTERFACE 4,2, (Summer 1982).

2. Chrisman, Carol, and Gerry Chrisman, "Retraining Faculty from Other Disciplines: An Alternate Source of Teachers," INTERFACE 4,2 (Summer 1982).

3. Codespoti, D. J., and J. C. Bays, "The University of South Carolina Computer Science Institute," SIGCSE BULLETIN 12,1, (February 1980).

4. Harrow, Keith, "A Faculty Development Program," SIGCSE BULLETIN 14,1, (February 1982).

5. Kroenke, David, "A Place in the Sun," INTERFACE 3,1 (Spring 1981).

6. Parker, Charles, "A Conversion Kit for Migrators to Computer Information Systems," INTERFACE 3,1 (Spring 1981).

7. Sharma, Onkar, and Ali Behforooz, "An Accelerated Program in Computer Science," SIGCSE BULLETIN 14,1 (February 1982).

8. Stegner, Richard, "Retraining: At the Graduate Level," INTERFACE 4,4, (Winter 1982-83).

9. Zientara, Marguerite, "University Summer School Retraining College Professors To Teach Computer Science," COMPUTERWORLD May 3, 1982.

Appendix

The following materials are representative of the requirements of the summer masters program at the University of Evansville.

Graduate Curriculum

First Summer Quarter

CS 501 SYSTEMS ANALYSIS 3 hrs. Discusses the complete role and functions of Systems Analysis. Considers the ten major steps in systems analysis from feasbility study to implementation. Considers the tools of the analyst in achieving a successful systems development. (Summer I)

CS 502 COMPUTER HARDWARE, 3 hrs. Covers the electronic and mechanical components of a computer, including processing units, memory units and input/output devices. Studies typical system configurations for various types of applications. Emphasis will be placed on typical systems in an educational environment. (Summer I)

CS 503 DATA STRUCTURES AND PROGRAMMING, 4 hrs. A modern introduction to programming techniques emhasizing structured style in PL/C and PL/I. Presents a broad exposure to algorithm analysis and implementation. Topics include stack and queue manipulations, tree travesal, recursion, search and sorting techniques, and basic file organizations. (Summer I and Summer II)

CS 504 SYSTEMS SOFTWARE, 3 hrs. Surveys Systems programs such as loaders, linkage editors, assemblers, compilers and operating systems. Covers the major components of each as well as design and implementation considerations. Emphasis will be placed on software available at students' home institutions. Prerequisite: CS 501, 502; Co-requisite CS 503. (Summer II)

CS 505 UNDERGRADUATE COMPUTING CURRICULA, 3 hrs. Discusses the curriculum movements and the model curricula in computer education. Considers the relationship of computing studies to liberal arts and develops guidelines for institutionally appropriate curricula. Prerequisite: CS 501 and 502; Co-requisite CS 503. (Summer II)

Academic Year

CS 510, 511 PRACTICUM, 3 hrs each. A supervised teaching experience in the discipline which will give the student opportunity to experience and analyze the unique characteristics of teaching computing. A course design and in-depth evaluation is required for each of the two registrations. Prerequisite: CS 505

Second Summer Quarter

CS 571, 572 COMPARATIVE PROGRAMMING LANGUAGES, 4 hrs. each. This two-quarter course introduces the principles of programming language design and implementation. The problems of automatic translation and the syntatic features of a variety of modern programming languages are examined. Emphasis is placed on finding a unifying perspective on programming languages which relates the general and special purpose languages as well as the high- and low-level languages. Concepts of teaching languages will also be presented. Prerequisite: CS 504 and 511. (Summer I and Summer II)

Second Summer Electives

CS 506 PROGRAMMING MICROCOMPUTERS, 4 hrs. A second course in BASIC programming which will concentrate on the graphics and data handling facilities available on common microcomputers.

CS 521 SOFTWARE ENGINEERING, 4 hrs. Presents the techniques of large-scale software development: project management and scheduling, unit and system testing, documentation and performance evaluation. Prerequisite: CS 501 and CS 503.

CS 551 INTRODUCTION TO MICROCOMPUTERS AND LOGIC DESIGN, 4 hrs. This course introduces the student to the major concepts in logical design of digital machines. Combinatorial logic is reviewed. Sequential design of digital machines (from counters and registers to microcomputers) is covered in depth. This course emphasizes the architecture system including the CPU, memory, and I/O. Real time software problems and sequential logic design lab problems are assigned. Prerequisite: CS 502 or CS/EE 350 or consent of instructor.

CS 599 INDEPENDENT STUDY, 1-4 hrs. Independent study of a topic or problem in Computer Science not otherwise covered in the curriculum. Prerequisite: permission of the faculty sponsor.

Also available are courses from other graduate departments, with permission of the program director.

REPRESENTATIVE COURSE MATERIALS

COMPUTING SCIENCE 501

COURSE TITLE: Systems Analysis

CREDIT:	SECTION	TIME/DAY	ROOM
3 hours	1	8:30-10 MTWT	?

COURSE DESCRIPTION: Discusses the complete role and functions of Systems Analysis. Considers the ten major steps in Systems Analysis from feasibility study to implementation. Considers the tools of the analyst in achieving a successful system development. Considers the pedagogics of the subject.

TEXT: Gane & Sarson: Structured Systems Analysis
Enid Squire: Introducing Systems Design

INSTRUCTOR: J. Westfall 479-2655

COURSE OBJECTIVES: Upon completion of this course the student should:

1. Understand the various roles of the Data Processing Professional.
2. Understand the need for structured analysis.
3. Be able to apply the tools of analysis.
4. Derive a physical system disign from a logical model.

EVALUATION: Evaluation will consist of two quizzes (50%), one final exam (25%) and two out of class assignments.

ASSIGNMENTS:

June 14	Chapters 1, 2 & 3	Squire	
15	Chapters 4 & 8	"	
16	Chapters 9 & 10	"	
17	Chapters 11 & 12	"	
June 21	Chapters 13 & 14	"	
22	Chapters 16, 17 & 18	"	
23	Quiz #1		
24	Chapters 1 & 2	Gane & Sarson	(Seminar Chamber of Commerce)
June 28	Chapter 3	" "	
29	Chapter 4	" "	
30	Chapter 5	" "	
July 1	Chapter 5	" "	
July 5	Chapter 6	" "	
6	Chapter 6	" "	(Homework #1 due)
7	Quiz #2		
8	Chapter 7	Gane & Sarson	
July 12	Chapter 8	" "	
13	Chapter 9 & 10	" "	(Homework #2 due)
14	Problems & lessons learned in teaching subject		
15	Final exam		

Summer, 1982

CS 502
COMPUTER HARDWARE

Instructor: Bruce Mabis

Text: "Computer Organization," Hamacher, Vranesic and Zaky, McGraw-Hill, 1978

Course Structure: Lecture and discussion (hopefully). Feel free to ask questions about the material as we discuss it.

Grading:	
4 Quizzes (25 pts each)	100 points
Final	100
2 Projects (50 pts each)	100
Assignments	50
TOTAL	350 points

Schedule: (Tentative)

Week	Topic
1	Chapters 1, 2, 3 Quiz 1
2	Appendix A and Chapters 4, 5 Quiz 2
3	Chapters 6, 7, 8 and Apppendix C Quiz 3
4	Chapters 9, 10 Quiz 4
5	Chapters 11, 12 Final

CS 503 SUMMER 1982

INSTRUCTOR:

Dr. William Mitchell (ES270: 479-2649)
Office 4-5 daily

TIME/PLACE:

ES164, 2:30-4:00 p.m. MTWTh (for now, will meet earlier second 5 weeks).

TEXT:

Tremblay & Sorenson, An Introduction to Data Structures with Applications (McGraw Hill 1976)

Coverage:		
Week 1:	Chapters 1 & 2	A reading guide will be supplied along with a set of study questions which will filter out the concepts of importance to the course.
2:	Chapter 3	
3:	4	
4 & 5:	5	
6 & 7:	6	
8 - 10:	7	

OBJECTIVE:

This course is intended to present the core concepts of programming as viewed by Computer Science. It will simultaneously address the specification and analysis of algorithms, the variety of conceptual data organizations, their impact on algorithms and their storage representation in computers, and the application of these data organizations in programming, emphasizing elegance in a rich and powerful programming language (PL/I). To achieve these goals we will write a lot (8-10) of programs and reflect upon the choices we make in the process.

PROCEDURE:

Monday's session will be devoted to explaning PL/I. The sessions of the remaining 3 days will be divided into 3 portions: a) a lecture expending or supplementing the assigned reading (maximum of 30 minutes); b) discussion of the study questions (not to extend past the first hour of class; and c) discussion of the programming problems. Great care will be taken to assure that each portion receives a full 30 minutes if that is perceived as desirable.

GRADING:

The course grade will be based on 9 sets of homework, 9 programs and a comprehensive final. Homework will be due each Monday. During the sixth week of class a midterm grade will be established and discussed in a scheduled individual ad vising session.

CS 503 QUESTION SET #1

1.) Write an algorithm of the proper form for converting a number (less than 4000) from Roman to Arabic.

2.) On our IBM 4331 a word is 32 bits. Assume a location in memory has the pattern 25736000000_8:

a.) If this is a two's complement interger value, what is its decimal equivalent?

b.) If it is a packed decimal integer value, what is its decimal equivalent?

c.) If it is a floating point excess 64 notation (7 bit exponent, 1 bit sign, 24 bit fraction), what is its decimal equivalent?

You see why a given location should only be allowed to have a single type.

3.) Find out the rules which govern the precision of the result when two fixed variables with precision (p_1,q_1) and (p_2,q_2) ($p_i > = q_i$ respectively) are combined arithmetically (+, -, /) in PL/I.

4.) Answer exercise 2-3.6.

5.) The KWIC_FORM algorithm on page 145 has some embarrassing errors - explain what they are. The KWIC_OUT and KWIC_FORM algorithms are inconsistent. Why? The strategy used to form these KWIC index is inefficient because you create a list of the unique key words and the process the duplicates. Examine the advantages of creating a list of all occurrences of key words including duplicates.

6.) Do problem 2-5.4.3 (Note it cannot be run in PL/C - Find out why.)

CS 503 PROGRAM 4

The central theme of this course is that we must become adept at modeling problems in computers, not by coding the problem into the limited data types implemented in the machine (integer and character in most, floating point in some) or provided by the programming language (arrays or structures and sometimes string facilities) but by utilizing the available data types and organizations to build the data structures which conform to the problem's requirements. It is fundamental that we not distort the problem when we bring it to the computer, but that we enhance the computer's ability through our software.

This theme has been developed by investigating a series of problems which involve manipulating high precision numbers. Note first that the IBM machine, and COBOL, provide high precision (15 digits) automatically, so that programmers will not need to be skilled in using data structures. Thus, the need for high precision numbers is not the justification for these illustrations (however, mini and microcomputer users need these techniques). Rather, the high precision numbers happen to be simple to present and understand, and thus provide an easily accessible vehicle for illustrating the theme.

Our first program illustrated programming the operation of division rather than using the built in operation and simulating high precision by controlling the format of the printed output. The second program utilized the technique of "array numbers" to gain an internal representation of numbers with high precision. The homework for Chapter 3 develops the concept of manipulating this representation. The relationship between the numeric and character forms of these numbers was developed as part of the problem of presenting these numbers on output.

The fourth program exercises using simply linked lists and also illustrates a more flexible store structure for multi-location numbers. There may be little practical need for linked list representation since most "real" situations are more efficiently handled with array numbers. But out goal is to become familiar with the tool rather than optimally solve a particular problem. The linked list requires greater concentration on the mechanics of supplying storage management facilities, as well as software implementation of the arithmetic operations. Therefore, the linked list structures illustrate more clearly the programmer's ability to construct the environment which best relates to the problem, rather than depending on the data structures available in the language.

CS503 Problem Set 5

1. Reverse the links on a general list with headers (write an algorithm or a program).

2. Trace the tordological sort algorithm in Knuth (use his data and verify his diagrams).

3. Do problem 1 on p. 412.

4. Do question 1 for a singly list list before you try it for the general list.

CS 503 NOTES FOR CHAPTER 5, PART I

SECTION	COMMENTS
5-1.1	The author approaches trees through directed graphs. The recursive definition given on page 313 is the more common starting point. Road maps are graphs, Pert harts are directed graphs. The discussion here is aimed at presenting vocabulary and illustrating each term with a picture. Most of the graph terminology is not important to us, but you should make a list of the italicized terms and write a definition for each in your own words. The tree vocabulary is more important (beginning on page 310). Be clear about level and degree. Become adept at interpreting any of the tree representation schemes (Venn diagrams, outline, parenthesis, prefix code, and first son-brother).
5-1.2	Learn each of the traversal sequences for an arbitrary binary tree. Study the PL/I figures. The algorithms use a stack. Study figure 5-1.20 and note the use of controlled storage to implement the stack. Study the trace for PREORDER given in table 5-1.1 where NA represents the address of the node containing A, and P is a pointer to some node on the tree. To visit P means to access the info stored at address P and output what you find. Threading is introduced on page 326. It uses the otherwise empty links in a tree storage scheme to avoid having to use a stack for traversal. You should trace the various traversals using the threads (dotted line in the example) and then see how a threaded tree is built following the algorithms. The material on representing forests (p. 328-332) can be passed over. Study, however, the examples of sequential storage of trees.
5-2	The applications section is once again heavily oriented toward compiler applications. Skip 5-2.1, 5-2.2 and 5-2.3. We will discuss in class much of the material in 5-2.4. Note that pages 351-355 merely define decision tables, while pages 355-361 discuss strategies for translating a given tableau (generating a tree representation). We will emphasize understanding the weighing assigned to the various paths through the tree which directs the choice in the Verhelst algorithm. The flow charts attempt to dramatize the different possibilities, but the examples on page 359 should be studied. We will skip the rest of the section beginning with p. 362.

Program 5

I have already distributed a handout which describes the algorithm for storing and traversing a network to determine the critical path. You assignment is to implement this algorithm in teams, one person in each team creating the subroutines LOAD, TRAVERSE, and CPATH, where LOAD reads the data stream and collects all the arcs and stores them in linked successor and predecessor lists with list pointers in the node table. TRAVERSE accesses these lists and computes the ETIME and LTIME for each node in the table. CPATH then computes the slack for each node and prints all the critical paths (zero slack). The interfaces between these modules may differ from team to team. However, PL/I will be used by everyone, the linked lists will be BASED variables, and the stack used in TRAVERSE will be implemented using a CONTROLLED variable. You should test your products singly and as a package before using my network: CS503P5

CS503 FINAL (take home part)

For each of the 8 programs assigned in CS503 respond to each of the following questions (record your responses on 8 sides of pieces of paper)

a. What PL/C (PL/I) syntax does this assignment teach (or might it introduce the student to) (Obviously the first assignment introduced quite a bit, successive assignments only incrementally increased the syntax knowledge. You may respond to the question either personally (I learned ....) or you may respond pedagogically (this assignment was intended to introduce the student to .....). Be obvious which position you take.)

b. What generalizable programming techniques did the assignment introduce? (What in the exercise would be useful in writing programs in another language--i.e. a syntax independent code-level technique)

c. What does this exercise (by virtue of its placement in the course or otherwise) teach about software development? (What professional programmer characteristics does it develop)

d. What, if any, algorithm does the assignment require the student to develop? (Identify the problems which were ultimately left to the student to solve and distinguish them from the algorithms which were fully described in the assignment specification.)

e. What are the opportunities allowed the student by the characteristic of the assignment to develop original and unpredicted approaches to solving the problem (if any).

f. What did the assignment teach the student (or require the student to find out) about users (the way they use programs or the content of their problems)? (What does the assignment teach about subjects other than programming)

Not all of the eight programs necessarily exhibit all six characteristics, so for those one or more of the subsections of your response will be "none". Certainly a student could blunder arround in solving an assignment and learn a great deal which was unintended, but useful. Rather than try to anticipate everything that might possibly be encountered, or all the things you learned extraneous to the assignment, indicate only those things which were probably learned from predictable errors, mindful that not everyone actually makes all the errors which the assignment tempts them to make.

CS503 Program #7

The records of the input file DATAFL are as described in the handout BUILD. In addition, a parameter card is read from SYSIN containing (in free format, no quotes) one of the following pairs:

INDIV, ZIP	(case 1)
INDIV, SHARES	(case 2)
GROUP, ZIP	(case 3)
GROUP, SHARES	(case 4)

where the comma may be omitted, and ZIP and SHARES represent numbers (zip code segment orshare numbers). For each input pair a particular set of address labels are extracted from the file:

case 1: separate label for each name in the address group having matching characters with ZIP beginning on the left end.
case 2: separate label for each name whose share amount equals or exceeds SHARES
case 3: single label containing as many names as the label will allow (in decreasing order of shares held) for each address group matching ZIP
case 4: single label containin, as many names as the label will allow (in decreasing order of shares held) where each name equals or exceeds SHARES

The generation of address labels requires careful positioning of the printer. Begiming at the top of page assume 6 lines per label and one line for spacing break. Labels may be generated in a column or three-up (assume 40 characters or width with one column separating). The file is kept LAST,FIRST so the name must be reordered (theoretically, the first or middle name might need to be truncated in order that the entire last name fit on the label, but this is quite unlikely given our name records).

The program is to be written in PL/I and will utilize the following file processing features: LOCATE mode of READ in order to recognize the record type,
RECORD output to printer (build an edited line image in a structure and WRITE the structure to the printer)
CLOSE the input file DATAFL at the end of the run, then loop back in a data-driven loop to read another input card and subsequently OPEN the file DATAFL for the generation of a second list of labels.
Turn in a run using the data in CS503P7

I encourage you to work in teams to implement a MAIN program which calls the following subroutines:

NEXT_LABEL(LINES,OK) which appropriately fills a 6-element character array or else turns the flag to '0'B. (CASE,SHARES, ZIP global)
NEXT_GROUP(OK) called by NEXT_LABEL to refresh from DATAFL the arrays holding the current address group.

The main program would read the parameter card, determine the value of CASE, and print the array LINES if it is OK. A single program will be submitted by a team, and the internal documentation will indicate the authors responsible for each segment.

CS505 1982 Syllabus

Text: Collection of Readings and CS105 Text

Requirements:
I. Course design for one of the courses you will teach this academic year.
create: syllabus (learning objectives, schedule)
final exam
due 8/4 bibliography (including potential or actual texts)
worked assignments (2-10 as appropriate: programs, problems, projects)
evaluation instrument
II. Syllabus of second course
due 8/11 sketch of assignments
III. Computer curriculum paper (evaluation of your school's present and/or
due 8/18 projected computing curriculum)
IV. Class presentation (discussion of your course) (schedule to be announced)
V. Midterm exam (8/9) over readings (details later)

Schedule: 7/19: Reflections on teaching computing.
20: The UE computing curriculum
21: ACM and IEEE Computer Science curriculum models
22: DPMA and ACM information systems curriculum models
26: Small college curriculum models
27: The first course: general education
28: the service course
29: the major's course
8/2: defining a discipline
3: conceptual hardspots
4: course design, curriculum design
5: resources and evaluation
Presentations: (A small college curriculum for a computing major)
10: the second programming course
11: systems analysis
12: data and file organizations
23: computer systems
24: projects and electives
25: implementation

26: Evaluation of summer 1982, academic year planning.

Grading: Each of the 5 assignments will be scored on a maximum of 30,10,20,10,30 points, respectively. A bonus will be earned for class participation.

90 pts.	= A
80	= B
70	= C

Office Hrs. 10-11 daily

Objective: The purpose of this course is to provide you the disciplinary context out of which the experienced computer educator operates. You will be exposed to the issues and methodologies that currently concern the college instructor, and some history on their development. Computing has been taught at the undergraduate level for barely 20 years, and the readings span that period. Each class period is devoted to a particular thread. I expect that you will familiarize yourself with the articles indicated for each day and come to class ready to question and interpret the issues raised with the goal of understanding the relevance of that issue to both the computer disciplines broadly and yourself specifically as you begin to practice computer education. I will try to resist lecturing for the first three weeks other than to begin each class with a brief statement as to why I choose the articles I assigned you to read.

Midterm Study Guide CS505

I. No questions on chapter I.

II. You should be prepared to discuss four kinds of curricula: Computer Science, (Two-Year) Computer Programming, Information Systems (CIS), Small College. You should be able to present an outline of the core courses or topics in each curricula and to present for each its distinguishing characteristics. You should be able to identify the different perspectives of the curricula committees and relate that perspective to the resulting recommendations. You should know the chronology of each recommendation movement.

III. You should be able to characterize and contrast each of three types of first courses: General Education (including introduction to Data Processing), service programming, and Major programming. The arguments for and against the unified first course should be understood.

IV. You should be able to outline the debate between theoretical and applied viewpoints on computing, and specifically, the arguments surrounding the mathematics component of computing education. You should also be prepared to discuss the implications of the accreditation committee reports in terms of similarities and differences, and future standardization of degree programs.

V. You should be able to discuss the problems presented to CS educators by the attitudes and abilities which students bring to the classroom in the first course. You should also be familiar with the variety of suggestions offered concerning the teaching of elementary programming (here and in chapter III) to the extent of relating those techniques to the attitude or concept problems identified earlier.

VI. You should be prepared to discuss practices relating to course design, text selection, and the professional development of faculty in computing.

Of the 10-12 themes identified above, the exposition of four will be required on the midterm, without notes.