AD-A265 088

ADA-BASED SOFTWARE ENGINEERING: UNDERGRADUATE CURRICULUM DEVELOPMENT

FINAL TECHNICAL REPORT

Grant No. MDA972-92-J-1025

for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

February 1993

James W. Hooper
Principal Investigator

Akhtar Lodgher
Co-Principal Investigator

Department of Computer Science and Software Development
Marshall University
Huntington, West Virginia 25755
(304) 696-5424

TABLE OF CONTENTS

1. BACKGROUND AND INTRODUCTION

The following inset paragraphs, extracted from the proposal for this grant, summarize the planned efforts.

> We propose to plan, develop, and bring about the first offering of, a two-course sequence in software engineering, emphasizing Ada as design and development language. The courses will become part of an undergraduate computer science curriculum. The courses will (tentatively) be designated as:
>
> Software Design and Development with Ada (SDDA)
> Software Engineering (SE)
>
> In performing this research, we will have two (related) goals:
>
> (a) to improve the computer science curriculum at Marshall University by implementing this new software engineering sequence, and
> (b) to enable other educational institutions to benefit from the work performed, by packaging course materials and suggestions for introducing the courses.
>
> The first of the two courses will be a new course at Marshall University -- entitled Software Design and Development with Ada (SDDA). It is envisioned as a junior or senior level course, and will require as a prerequisite a course in concepts of programming languages, which at MU is CIS 320 Programming Languages. CIS 320 will be enhanced to emphasize Ada as an example language, and will require that at least one sizeable program be developed in Ada. In the early offerings of the new SDDA course, it will be necessary to emphasize the syntax and semantics of Ada, due to the expected lack of preparation of the students.
>
> The existing MU course, CIS 479 Software Engineering, will be revised and augmented to mesh with the new course SDDA, and especially to emphasize the life cycle phases not covered in SDDA (i.e., requirements determination, testing, integration, maintenance). The overall software process will be emphasized (products, methods, tools), with the benefits of Ada being stressed. The proposed courses will be project-oriented, and will emphasize a team approach to the solution of realistic problems.
>
> We would prefer to have freshman-level course(s) that are Ada based (i.e., CS 1, CS 2), which could be assumed as prerequisites to SDDA. Since it is not feasible to bring this about at MU at this time, we will devise and document a plan for adapting SDDA at a later time when freshman-level Ada courses are offered. This approach will, of course, emphasize less the fundamentals of Ada within SDDA, and place greater

emphasis on design and development, per se. This alternative plan could be implemented initially by other institutions, given sufficient resources.

The grant proposal was submitted on September 30, 1991, with the expectation that, if the grant were awarded, the period of performance would be January 1, 1992 through August 31, 1992. As it turned out the grant award was for a period of performance from June 1, 1992 through February 28, 1993. In the interim between September 30, 1991 and June 30, 1992, some very significant events occurred at Marshall University that caused us to take a different approach than we had originally anticipated. We have achieved a great deal more relative to computer science and software engineering education at Marshall University than we could have anticipated. We have, in fact, completely revamped the B.S. degree program in computer science, instituting a strong emphasis in software engineering in the new curriculum (including teaching Ada as the introductory language at the freshman level). We are reporting the details of that work in this Final Technical Report, as well as the specifics of our planned software engineering courses. Consequently we believe that the results we are reporting will be of much greater value to other computer science/software engineering educators than the exclusive focus on two software engineering courses we had originally planned.

At this point we will briefly review the events and planning that led to our new curriculum. Full details of the planning as of June 1992 are reported in a paper prepared and presented by James Hooper at the SEI Software Engineering Conference in San Diego, CA in October 1992 (provided in Appendix A) [Hooper92]. The curriculum is summarized in section 3.

The Department of Computer and Information Sciences (CIS) was within the College of Business at Marshall University (M.U.) until Fall 1991. It offered a B.S. degree with emphasis in information systems (business-oriented), as well as a degree emphasizing "main stream" computer science. The B.B.A. degree in the College of Business also included an option in Business Information Systems (B.I.S.), with the computing-oriented courses being offered by CIS. The "main stream" B.S. was patterned after the 1978 ACM curriculum, but was lighter in science and mathematics. The information systems option was lighter still in science and mathematics, and required a different set of computing courses at the junior and senior levels. The B.I.S. had less emphasis on computing courses, and greater emphasis on business courses. There was some concern among area employers about the capability of some of the graduates.

In September 1991 the CIS Department was moved to the College of Science, and was tasked by the M.U. President to update and strengthen the curriculum, and emphasize software engineering. The business-oriented information systems option was dropped from the B.S. degree program. At the time of this Final Report the Management Department within the College of Business has proposed a new curriculum for the B.B.A. in B.I.S. Concurrent with moving the Department, the PI of this grant was designated Chairman of Computer Science Transition Planning; the Co-PI was very much involved in the curriculum planning also. An Advisory Panel for Computer Science was formed, consisting of senior computing managers from the region. The overall objective of the effort was to develop a strong academic computer science program, with software engineering emphasis, whose graduates would be well prepared to function as computing professionals in a diversity of organizations. The paper in Appendix A includes considerable detail on the dynamics of the curriculum design process, including the role of the Advisory Panel.

In September 1992 the M.U. Faculty Senate gave approval to the new curriculum. Appendix B contains the Catalog descriptions of the new computer science courses, as well as a summary of all requirements for the degree, and syllabi for selected courses. Also, in Fall 1992 the name of the department was changed to The Department of Computer Science and Software Development (CSD), to better convey the balanced emphasis on both computer science and software engineering.

We provide some observations on the new curriculum in section 3. In section 2 we give a discussion of our rationale for selecting Ada as the introductory teaching language at the freshman level. Section 4 provides a discussion of the software engineering courses in our new curriculum. Section 5 discusses alternative approaches that may be taken to the preparation of students in the Ada language within a computer science/software engineering curriculum. Finally, section 6 offers some summary conclusions and recommendations.

2. THE ROLE OF ADA IN AN UNDERGRADUATE CURRICULUM

We wish to give a brief discussion of our rationale for selecting Ada as the introductory teaching language at M.U. A great many computer science departments are making changes in their teaching language at this time. Pascal has been very popular in past years, but is waning at the present time. At M.U. Pascal had been for many years the introductory language. Pascal provided the best available alternative earlier-on, providing a structured syntax and enforcement of good programming practices. At the present time Ada offers more desirable syntax and semantics than Pascal, and is a "marketable" language for professional employment, whereas Pascal is a "dead end" in that respect. We also consider Ada/PDL attractive for use in requirements and design specification. In summary, Ada is in our opinion much more supportive of effective software engineering practice than any other currently-available programming language.

Many colleges and universities are changing to C or C++ as their teaching language, with a few choosing Modula 2 or some other. We do not consider C or C++ to be good introductory languages, even though they are very important in government and industry. There is in fact a much greater demand currently for college graduates who are familiar with C than there is for Ada. We thus feel an obligation to provide considerable exposure of our students to C and C++, but we prefer to have them learn C and C++ after they have first learned good programming practices with Ada.

Thus we began the use of Ada in our freshman courses in Fall 1992. We offer specific discussion about our approaches to Ada in the following sections.

3. OVERVIEW OF A B.S. DEGREE PROGRAM IN COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Appendix B contains full documentation on our new B.S. degree program in Computer Science and Software Development. We had the following goals in arriving at the curriculum:

- Meet the CSAB quantitative requirements
- Meet the Computing Curricula 1991 CS requirements
- Have a strong software engineering emphasis
- Provide students practical experience participating in:
    - Team-oriented software projects
    - Development and maintenance of large software systems
    - Formal treatment of life cycle activities--including creation and documentation of requirements, designs, and code, participation in formal reviews, change control boards, ...
- Place strong emphasis on the "systems approach" to software development, beginning with the very first freshman-level course
- Provide background and experience in systems engineering as it relates to software--i.e., embedded systems, hardware/software tradeoffs; simulation, queuing theory, for use in determining requirements feasibility, performance evaluation, etc.
- Emphasize management methods, cost analysis, quality assurance, decision support systems
- Emphasize supervised teaching laboratories as part of some courses

We believe we have attained all these goals in the resulting curriculum. We have emphasized oral and written communications skills, and require a strong foundation in mathematics. We began the new courses in spring semester 1993, and will continue to phase in the new courses through spring 1995. The following section describes the software engineering courses in the new curriculum.

4. THE SOFTWARE ENGINEERING COURSES

As we have noted in earlier sections, from the first freshman course we are emphasizing software engineering in the new curriculum. An orientation to the software life cycle is given in the first freshman course. Of course, some courses deal exclusively with software engineering. There are four such courses, constituting a total of 12 semester hours. Appendix B contains syllabi for these courses, which are as follows:

CSD 313 Introduction to Systems and Software Engineering
CSD 333 Software Engineering
CSD 493 Senior Team Project Sequence, First Semester
CSD 494 Senior Team Project Sequence, Second Semester

CSD 313 and CSD 333 correspond approximately to the two courses we proposed to develop in our grant application (having suggested names Software Design and Development with Ada, and Software Engineering). Since we were totally redesigning the curriculum, we had the benefit of being able to design the courses directly into a new curriculum, rather than trying to "patch up" existing courses.

CSD 313 emphasizes the role of systems engineering methods in requirements determination, including embedded systems, and emphasis on maintenance. We intend to present the evolving object-oriented view of requirements specification, as well as the more mature function-oriented view. CSD 333 emphasizes design, both object-oriented and function-oriented. Software reuse is emphasized, as well as configuration management, verification and validation, and other issues within the software process. Again, the syllabi in Appendix B provide further details of these courses.

CSD 493 and CSD 494 together constitute a full year's experience in carrying out an entire system and software engineering life cycle for a project of realistic size and complexity. Our intention is that there be a "real" problem, and a "real" customer, with formal documentation and reviews. While much of this experience will be project oriented, we will also present new material in lectures that time would not permit in earlier courses, paced to be useful in the project work.

5. APPROACHES TO STUDENT PREPARATION IN ADA

As our curriculum now stands Ada is taught at the freshman level, so students coming into our software engineering courses are already Ada-literate. In this section we discuss our methods and experience in introducing Ada at the freshman level. For the benefit of educators who may not be able to teach Ada to their students prior to their participation in software engineering courses, we also discuss the approach we took prior to changing from Pascal to Ada in our freshman courses.

Ada was first taught by Professor Lodgher (the Co-PI) in the Fall semester 1991 as part of the course CIS 320, Programming Languages. Ada was used as an example language to emphasize modular programming, incorporation of safe programming principles, embedded programming and for designing large software systems. Ada was then taught as a separate course at the senior level in the Spring semester 1992 by Professor Lodgher (see Appendix C - paper presented at the SESCCC 92 on how to teach Ada at the senior level [Lodgher 92]). By this time the new curriculum was in place and Ada was to be taught at the freshman level from Fall 1992. Thus the freshman sequence of courses (CSD 119 - Introduction to Computing I, and CSD 120 - Introduction to Computing II) to be taught using Ada as the programming language, had to be developed.

It was decided that each of these courses should have a mandatory closed lab component. The courses were designed to be 4 credit hour courses: three hours of in-class lecture with 2 hours of closed lab work. Part of the summer of 1992 was used in developing the materials for the first course (CSD 119). The materials include the preparation of detailed syllabus, choosing the text book, designing the programming exercises, designing the laboratory manual, the syllabus, and class assignments (see Appendix D). The laboratory manual (*"Computer Programming I - Laboratory Manual"*, by Akhtar Lodgher) is being submitted to DARPA as a separate deliverable. The major emphasis of the grant proposal was to impart software-engineering training in Ada and thus it was decided that the materials of the course CSD 119 be prepared with meticulous care. This course was offered in Fall 1992 and many innovative ideas such as completion of design before code, use of teams, etc., were used in the course (see Appendix E which contains a copy of [Lodgher 93]).

The hardware facilities used for programming was a lab of about 30 VT200+ terminals connected to a DEC VAX machine. The DEC Ada compiler running under the VMS operating system was used. The university has recently opened a new laboratory equipped with about 25 486-based microcomputers, networked to a server. Currently an Ada compiler is unavailable on this network, but the university is planning to install one shortly. Another laboratory, equipped with about 25 386 machines (without hard disk) and networked to a slightly slower server is also available. It is anticipated that a CASE tool will also be made available on these machines.

Prof. Lodgher has applied for an National Science Foundation (NSF) Instrumentation and Laboratory Improvement (ILI) grant to equip a laboratory with six networked Sun SPARC workstations. The grant includes the cost of installing an Ada compiler along with a powerful CASE tool.

In Spring semester 1993, Prof. Hooper served as instructor during the last offering of CSD 479 (previously CIS 479). He also taught CIS 479 during Spring 1992. Some incremental changes to this course were made during these two offerings, with the development/phasing in of CSD 313 (Introduction to Systems and Software Engineering) and CSD 333 (Software Engineering) during academic year 1993-94 in view, and the planned discontinuation of CSD 479. The syllabus for the current (1993) offering of the course is shown in Appendix F, along with the team project assignments for the last two offerings of this course.

The changes made during the last two offerings included strong emphasis on Ada. The textbook (Ian Sommerville's SOFTWARE ENGINEERING, 4th ed., Addison-Wesley, 1992) was chosen in large measure because of its emphasis on Ada (including an Ada-based program description language (Ada/PDL)), in addition to its readability and comprehensiveness in treating modern software engineering issues. This textbook provides an appendix which introduces Ada, comparing it with Pascal. The appendix also summarizes an Ada/PDL design example. The instructor spent about one class hour summarizing this material for the benefit of those who had not yet been exposed to Ada--which was most of the class members. Most of the class members had previous Pascal experience, which was very helpful. The book chapters on requirements specification and object-oriented design make use of Ada/PDL as the basis for examples. The book emphasizes software reuse, including a separate chapter on the topic. Ada is used in this material as well to illustrate and explain reuse throughout the life cycle. Thus the book's orientation fits well into our overall strategy to emphasize Ada throughout our curriculum.

As was mentioned above, Appendix F shows the team projects for the last two offerings of CIS/CSD 479. In both of these cases the teams were encouraged to make use of Ada/PDL in determining requirements and high-level design.

For departments that are not yet in position to extensively revise the curriculum to emphasize software engineering, but who wish to give some exposure to Ada throughout the life cycle in a single junior/senior level software engineering course, the approach taken in CIS/CSD 479 has proven by experience to work out rather well. It would be desirable to carry out the high-level design developed in such a course, into detailed design and implementation by use of Ada. Perhaps some departments may wish to consider a follow-on course which emphasizes implementation with Ada. In that case the instructor may wish to make use of self-paced computer-assisted tutorial assistance such as is offered by NSITE Ada, since class time available for Ada discussion will likely be very limited.

In our revised curriculum, CSD 313 and CSD 333 together will accommodate the entire life cycle, although CSD 313 will not attempt to cover as great scope of material as CSD 479 has attempted. And, as mentioned in the previous section, the senior team project sequence will provide further, even more intensive experiences in life cycle activities, including Ada as implementation language.

6. CONCLUSIONS AND RECOMMENDATIONS

We have been privileged to completely redesign our undergraduate computer science curriculum, which no doubt is a rare opportunity for any academic department. We have taken advantage of this opportunity to make software engineering an important, integral part of the new curriculum. At the same time we have chosen to begin the use of the Ada language as the introductory language, and as the primary language in our software engineering courses. We expect these decisions to result in graduates who are well prepared to take their place as professional computer scientists and software engineers. We have much work remaining as we phase in our new courses. CSD 313 and CSD 333 will be taught for the first time in academic year 1993-94, and CSD 493 and CSD 494 will be taught in academic year 1994-95.

There is no doubt that it is preferrable to be able to study Ada prior to taking software engineering courses, as is now the case in our new curriculum. However, our experience has shown that advanced students with a good background in programming can learn enough about Ada while taking a software engineering course to understand how to use Ada/PDL in requirements and design specification. A follow-on course in implementation with Ada would be very desirable, giving an opportunity to gain significant exposure to Ada. Use of such tutorial assistance as is provided by NSITE Ada could help overcome the lack of exposure to Ada earlier in the program.

We are pleased to be able to provide the results of our curriculum redesign as part of this submission. We are appreciative for the support from DARPA in this effort, and believe the grant has resulted in greater leverage than either DARPA or we could have anticipated.

REFERENCES

[CC91] Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force; ACM Press/IEEE Computer Society Press.

[CSAB] 1990 Annual Report for the year ending September 30, 1990; Computing Sciences Accreditation Board, Inc.

[FORD90] 1990 SEI Report on Undergraduate Software Engineering Education, CMU/SEI-90-TR-3.

[Hooper92] Hooper, James W., *"Planning for Software Engineering Education Within a Computer Science Framework at Marshall University"*, in Proceedings of The SEI Software Engineering Education Conference, San Diego, CA, October 1992.

[Lodgher92] Lodgher, Akhtar, *"Teaching Ada at the Senior Level"*, in Proceedings of the Sixth Annual Small College Computing Conference, Jefferson City, TN, Nov 1992.

[Lodgher93] Lodgher, Akhtar, and James Hooper, *"Using Ada For a Team Based Software Engineering Approach to CS1"*, in Proceedings of the 11th Annual National Conference on Ada Technology, Williamsburg, VA, March 15-18, 1993 .

APPENDIX A

"Planning for Software Engineering Education Within a Computer Science Framework at Marshall University"

C. Sledge (Ed.)

Software Engineering Education

SEI Conference 1992
San Diego, California, USA, October 5-7, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Planning for Software Engineering Education Within a Computer Science Framework At Marshall University

James W. Hooper

Department of Computer and Information Sciences
Marshall University
Huntington, West Virginia 25755
Telephone: (304) 696-2693
Fax: (304) 696-4646
email: CIS010@marshall.wvnet.edu

Abstract. This paper presents an overview of the planning process undertaken at Marshall University, to update and strengthen the undergraduate computer science program, and to introduce a strong emphasis in software engineering. An Advisory Panel of senior computing managers from the region was formed, and has provided valuable input to the "requirements determination" phase of the planning process. Based on the Panel's input, and guided by current curriculum design recommendations and accreditation standards, the computer science faculty--with the participation of other colleagues--has undertaken a total revision of the curriculum, addressing other issues as well, such as laboratories and staffing. Summaries are provided of the Panel's input, and of the current draft status of the evolving curriculum.

1 Introduction

Marshall University has for some years offered the B.S. degree in computer science, with emphasis in either information systems (business-oriented) or "main stream" computer science. There has also been a Business Information Systems (BIS) major in the BBA degree program. The Department of Computer and Information Sciences (CIS) has been within the College of Business since its creation as a department. The main stream computer science degree was patterned along the lines of the 1978 ACM curriculum, but did not emphasize science courses, and did not strongly emphasize mathematics. The information systems degree has been lighter still in mathematics emphasis, and has required a different set of computing courses at the junior and senior level. The BIS program has placed less emphasis on computing courses and greater emphasis on business courses.

The relationship of the department to the College of Business and other business departments has often been uneasy at best. It has been difficult to bring about acceptance of strong science and mathematics emphasis among business-oriented students, and there has often been a mismatch of viewpoints and expectations between CIS faculty and other College of Business colleagues. The teaching loads have been heavy (nominally twelve hours per semester), and it has thus been very difficult for the faculty to conduct research and publish. A significant manifestation of these problems has been the high turnover of CIS faculty. Also, there has been a concern among some area employers that some graduates of these degree programs were not able to be productive soon after employment.

Discussions were held on campus during the summer of 1991 concerning a possible move of the CIS Department to the College of Science. Both of the Deans (Business and Science) and the Provost came to agree that the proposed move was likely to benefit the students in the CIS programs. The transfer became a reality in late September, when President J. Wade Gilley--newly-appointed president of Marshall University (MU)--wrote a letter approving the transfer of the Department and programs to the College of Science. Dr. Gilley came to MU from George Mason University (GMU) where he was Senior Vice President; an engineer by education, he was very much involved in the growth and development of GMU, including engineering programs, computer science, and software engineering. He made known his desire that the transfer of CIS should include curriculum updating and strengthening, and that software engineering should be emphasized due to the needs of the region. The author came to MU in late August of 1991 as a visiting professor--on leave-of-absence from the University of Alabama in Huntsville. The position (The Arthur and Joan Meyer Weisberg Chair in Software Engineering) was endowed by a successful civic-minded couple with business connections to a Huntington-based software engineering firm, who were aware of the need to strengthen the computer science programs at MU. The author was appointed as Chairman of the Transition Planning Committee for Computer Science, tasked with "effecting an orderly transition of the department". The Acting Chairman of the CIS Department continued in that position, administering the on-going programs.

2 Planning Approach

The first step in the planning process was to form the Transition Planning Committee for Computer Science. The committee consists of all members of the CIS faculty (seven full-time faculty), the Associate Director of the Computer Center, an engineering professor active in planning for a manufacturing sciences degree program, and a management professor from the College of Business having a strong management information systems (MIS) background. It was also considered critically important to establish effective interrelationships with computing organizations/professionals in the region. To this end, a number of senior computing managers from the "Tri-State" area were invited to join the Computer Science Advisory Panel. Current members are from Ashland Petroleum Co., Inco Alloys International, Inc., State Electric Supply Co., Strictly Business Software Engineering, and Union Carbide Chemicals and Plastics Co., Inc. All of these managers have manifested a very strong interest in improving computer

science education at MU, and have actively and thoughtfully participated in strategy sessions. They have all had experience with MU computer science students, as interns/coops or regular employees. Monthly meetings of the Advisory Panel have been held beginning in November 1991; the Dean of the College of Science and the author have participated in all meetings. Numerous working sessions have been held by the faculty, at various times also including other Transition Planning Committee members.

It is understood that the faculty has "the last word" on curriculum planning. However, the faculty has welcomed the active participation of others both within MU and outside MU, recognizing the importance of their input in building and sustaining a high-quality academic program. It is the motivating objective of the faculty to develop a strong academic computer science program, with software engineering emphasis, whose graduates will be well prepared to function as computing professionals in a diversity of organizations. The opportunity to participate in achievement of this objective has been accepted with enthusiasm by the faculty.

The scope of activities necessary to achieve the overall objective includes more than curriculum, of course. The following list of necessary planning activities was developed, characterizing the scope/responsibilities of the Transition Planning Committee:

* Determine long-term instructional, research, and service goals for computer and information science programs at MU

* Plan revised B.S. curriculum, including emphasis on software engineering, and taking into account interrelationships with other existing and planned curricula and MU research thrusts

* Determine a (longer-term) plan for an M.S. degree program; take these preliminary requirements into account while planning the B.S. curriculum

* Determine resources necessary for the revised B.S. program

* Develop a transition plan for introducing the revised B.S. program(s) and phasing out the existing B.S. programs; this should include feasible schedules, and should document assumptions about availability of resources

* Determine a strategy for increasing research activities in the department, including specific research thrusts, and prospective funding sources

* Develop a strategy for faculty development, recruitment, and retention, based on anticipated program redirection and growth

Most of these activities are in progress at the time this paper is written. Greatest emphasis so far has been given to curriculum development, and most of the following discussion focuses on that activity.

3 Curriculum Planning

The Advisory Panel was requested to provide insight to MU concerning their viewpoint of the characteristics we should seek to bring about in the B.S. program in computer science. (I.E., we sought thei· input to the requirements determination process!) After discussion sessions and reviewing a previous draft, the Panel provided the Mission/Vision and Standards Statements shown as Figure 1.

MISSION/VISION

The Computer and Information Science (CIS) Advisory Panel was created to provide a vehicle for dialogue between the College of Science and the business community regarding Marshall University's Computer and Information Science program. The Mission and Vision of the Advisory Panel follows:

The Mission is:

- To provide leadership and technical expertise in identifying and communicating to the College of Science appropriate programs and curricula that will result in a highly developed Computer and Information Science graduate suitable for regional employment.
- To enhance the overall competitive position of Marshall University in attracting faculty, research grants and support of the business community for the CIS program.
- To suggest programs and strategies in accordance with our Mission that will accelerate the accomplishment of the Vision.

When the Mission is accomplished, it is our Vision that:

- The CIS program at Marshall will exceed student's expectations and graduates will be highly sought by the employers.
- Marshall University will have achieved a competitive advantage in attracting faculty and business support for the CIS program.

(continued on next page)

Figure 1. Advisory Panel Mission/Vision and Standards Statements

---

- Lasting alliances between the business and academic community will have been established in support of the CIS program.

STANDARDS

The standards, outlined below, are the definitive milestones that measure our progress toward the Vision.

- The CIS program at Marshall will be linked to the business community and be driven by requirements of regional enterprises.
- The CIS program will consistently provide high-quality graduates.
- The CIS program will be accredited by the appropriate accrediting organization(s).
- Partnerships will be established between the business community and the University to establish student work programs, faculty research projects and technology transfer.
- The University can attract and tenure professors in the field of computer science.
- The CIS program will attract top area students because of its reputation for excellence.

Figure 1. Advisory Panel Mission/Vision and Standards Statements (Continued)

We also asked the Panel to provide input concerning their viewpoint of the characteristics a computer science graduate should possess to be successful in the workplace. Their response is summarized in Figure 2. The author's own long-term experience with government and industry software projects leads to strong concurrence with the emphasis given by the Panel to personal communication skills, and also to project management considerations. In other words, technically-oriented college graduates need to be more than "techies"--they must be able to organize and communicate ideas, understand management principles, have a "system view" of development and maintenance, and understand principles relating to cost and trade-off analysis.

Based on the Advisory Committee's input, the faculty's knowledge of other computer science curricula (including the author's long-term participation in the University of Alabama in Huntsville's accredited B.S. program in computer science), and recent guidance on curriculum design (especially [CC91], [CSAB], [FORD90]), we undertook an extensive B.S. curriculum revision. We established the following (preliminary) list of goals for the B.S. program:

* Meet the CSAB quantitative requirements
* Meet the Computing Curricula 1991 CS requirements
* Have a strong software engineering emphasis
* Provide students practical experience participating in:
  * Team-oriented software projects
  * Development and maintenance of large software systems
  * Formal treatment of life cycle activities--including creation and documentation of requirements, designs, and code, participation in formal reviews, change control boards, ...
* Place strong emphasis on the "systems approach" to software development, beginning with the very first freshman-level course
* Provide background and experience in systems engineering as it relates to software--i.e., embedded systems, hardware/software tradeoffs; simulation, queuing theory, for use in determining requirements feasibility, performance evaluation, etc.
* Emphasize management methods, cost analysis, quality assurance, decision support systems
* Seek a synergistic relationship with Manufacturing Science
* Emphasize supervised teaching laboratories as part of some courses

COMMUNICATION
Oral Expression--Competent in:
Vocabulary use (general business, specific technical)
Speaking and presentation (including use of audio-visual aids)
Written Expression--Competent in:
Organizing and representing thoughts clearly
Using word processing or desktop publishing tools
Graphical representation and graphics tools
Listening

ANALYTICAL
Mathematics
Problem solving--Competent in:
The use of problem prevention and solution methodologies
The use of statistical analysis tools
Modeling
Problem decomposition, process flow, context, data
Decision theory

(continued on next page)

Figure 2. Advisory Panel Capabilities List for Success in a Working Environment

TECHNICAL
Data types, structure, organization, dictionaries,
languages, algorithms, operating systems,
hardware, communications,
standards, security, documentation,
CASE concepts, prototyping concepts

PROJECT
Competent in software engineering life cycle concepts:
analysis, design, construction, testing, implementation,
change management

PERSONAL
Personal behavior management
Meeting planning and leading
Group techniques and team processes
Organizational behavior

Figure 2. Advisory Panel Capabilities List for Success in a Working Environment (Continued)

Pascal had been for many years the introductory teaching language at MU. While it has served well in that role, it is a "dead end" relative to use in development of large software systems. Thus we decided to use Ada as the primary teaching language, beginning with an appropriate subset (essentially equivalent to Pascal) in the first freshman-level course. We consider Ada to be the best available language for encouraging good software engineering and programming principles. We believe that its use as the primary teaching language will help enforce and facilitate good practices from the beginning, and will also provide the graduates working knowledge of a language that can be used in professional practice. We also feel an obligation to ensure that the graduates are familiar with the C language, due to the demand by prospective employers, but believe it is best to defer its use until programming methods and skills have been established by use of Ada. We are strongly committed to provide concepts and experiences to the students different from the "toy problem" approach of most current computer science courses, which are usually solved by a student working alone. Many of us are well aware of the laments of employers of computer science graduates who must give new graduates extensive training before they are capable of performing as professional software engineers. This revised curriculum emphasis must include participation in developing and maintaining large software systems, following a well-defined software process, and working as part of teams.

Figure 3 summarizes briefly the current draft status of the evolving curriculum. We emphasize at this point that this draft material is presented to give a general understanding of our current curriculum emphasis and planning status, but that any of the specifics of the curriculum are subject to change as we proceed. We have proposed that the name of the Department be changed to reflect the increased emphasis on software engineering, while still reflecting the emphasis on a solid computer science base. The name under current consideration is "Computer Science and Software Development"; thus the course designations use "CSD" as a prefix. We are planning to delete all existing CIS courses; even though some are similar to replacement courses, we want to institute a new numbering scheme and impose different prerequisites in many cases.

We are planning no separate information systems option in this program; it is our opinion, supported by the Advisory Panel, that a good quality program with a software engineering emphasis should be applicable to business and management applications, as well as to engineering and scientific applications. We believe that all students, of all majors, should be exposed (if not already in high school) to word processing, spread sheets, databases, control statements, electronic mail, and some fundamental concepts about programming and a programming language. Exactly how each college at MU will approach such computer literacy is still under discussion.

We have included one course each in management, economics, and accounting, to help prepare the students for software engineering practice, and to help develop a working knowledge of business-related concepts and terminology for use in possible future computing applications. Speech and technical writing courses are included to help develop communication skills. Courses in physics and chemistry are specified to help develop a necessary foundation for industrial computing applications (numerous chemical industries operate in the *Tri-State region, for example*).

4 Conclusion

Much work remains, including completing detailed curriculum planning and implementing the new courses, upgrading laboratories, recruiting faculty, and strengthening research. We are planning to gradually phase in the new courses, beginning in the Spring semester of 1993, culminating in the offering of the Senior Team Project Sequence in academic year 1994-95. The first increment of the proposed curriculum has been reviewed and approved by college and university-level curriculum committees, and awaits approval by the Faculty Senate. We plan to report on further phases of the work as it proceeds. We are very much encouraged by the enthusiasm of the participants in this process, and by the strong support from the MU administration. Most of all, we believe that the real "winners" from this effort will be the young people of the region who graduate from the resulting program.

*************************** PLEASE NOTE: ******************************

THE FOLLOWING IS A BRIEF SUMMARY OF THE CURRENT STATUS OF THE PROPOSED CURRICULUM. SOME CHANGES MAY OCCUR AS DETAILED PLANNING IS COMPLETED AND APPROVAL REVIEWS OCCUR.

****

CSD 100 Fundamentals of Computing (i.e., computer literacy)

* 3 sem. hours: 2 hours lecture, 2 hours supervised laboratory
* CSD majors may not receive credit hours for this course

REQUIRED COURSES (38 Sem. Hrs)

CSD 119, 120 Introduction to Computing I, II

* PR: CSD 100 or equiv. knowledge, CSD 119, respectively
* 4 sem. hours each: 3 hours lecture, 2 hours supervised lab.
* Introduce the software life cycle, placing software design, coding, unit testing, in that context; reading, alteration of existing code; reuse; design/programming using Ada; data structures emphasis in 120. Team concepts, experiences; use of simple CASE-like tools. Social, ethical, professional issues
* see CD 101/102 in Computing Curricula 1991 (CC91)

CSD 212 *Introduction* to Computer Engineering

* PR: CSD 120
* 3 sem. hrs.: 2 hours lecture, 2 hours supervised lab.
* Essentially equiv. to "Switching Theory"
* See CD 201 in CC91

CSD 222 Computer Organization and Assembly Language Programming

* PR: CSD 212
* 3 sem. hrs.
* Register level architecture of a specific processor, assembly language programming for that processor; data representation, I/O devices, bus transactions
* See CD 301 in CC91

CSD 240 Analysis and Design of Algorithms

* PR: CSD 120, Discrete Structures
* 3 sem. hrs.

(Continued on next page)

Figure 3. Draft Curriculum Summary

* Includes advanced data structures coverage; intro. to object-oriented design; complexity, recursive algs., computability; relationship to software eng. / large software systems; continued use of Ada.
* See CD 202 in CC91

CSD 313 Intro. to Systems and Software Engineering
* PR: CSD 240
* 3 sem. hrs.
* The complexity of large systems; the systems approach; life cycle activities; methods and tools of systems engr.; requirements determination, trade-off studies, cost estimation; tools and methods of systems engr., quality assurance, human engineering
* The relationships of systems engr. and software engr.; systems engr. in embedded-systems applications; allocations to softw., hardware, communications, people; CASE tools (Computer-Assisted Systems Eng./ Softw. Eng.): e.g., RDD 100; management approaches, risk analysis
* The Softw. Engr. life cycle / software process
* Team project that provides experiences in sys. eng. combined with softw. engr.

CSD 322 Computer Architecture
* PR: CSD 222, CSD 240
* 3 sem. hrs.
* Design alternatives, instruction set architectures, memory organization, interfacing, alternative computer architectures
* See CD 306 in CC91

CSD 325 Intro. to Programming Languages
* PR: CSD 222
* 3 sem. hrs.
* Emphasis on programming language principles, illustrated by syntax and semantics of constructs in current languages
* Imperative and functional languages; concurrency; logic-based approach; object-oriented approach
* See CD 304 in CC91

CSD 333 Software Engineering
* PR: CSD 313
* 3 sem. hrs.

(continued on next page)

Figure 3. Draft Curriculum Summary (Continued)

* Further experience in requirements analysis and specification; functional and object-oriented design, user interface design, implementation in Ada, verification and validation issues, systems integration; maintenance; reuse; real time; safety; configuration management
* Team project experience, large software system
* See CD303 in CC91

CSD 338 Operating Systems
* PR: CSD 222, CSD 240
* 3 sem. hrs.
* Process management, device and memory management, security, networking, distributed operating systems
* File structures, UNIX emphasis, programming in C
* See CD 305 in CC91

CSD 493,494 Senior Team Project Sequence
* PR: CSD 333, CSD 322
* 3 sem. hours each
* A two-consecutive-semester course sequence providing a realistic experience in team-based software development. Student teams will perform requirements analysis and specification, design, implementation, testing and integration. In-class reviews will be conducted by the teams. The project is to be of significant size and complexity. Project management will be stressed, including costing and scheduling.
* See CG407,408, CC91

ELECTIVE COURSES
(A student will select three courses to complete major, totalling 47 sem. hrs.)

CSD 345	Software Development for Business/Management (COBOL)
CSD 356	Scientific Computing / Supercomputing (FORTRAN)
CSD 367	Systems Programming (C, Unix emphasis)
CSD 409	Computer Information Systems for Health Care (not available as part of major for computer science majors)
CSD 419	Decision Systems
CSD 429	Intro. to Computer Graphics
CSD 439/539	Intro. to Artificial Intelligence

(continued on next page)

Figure 3. Draft Curriculum Summary (Continued)

CSD 442/542	Communic. Networks and Distributed Systems	
CSD 449/549	Formal Languages and Automata Theory	
CSD 457/557	Database Systems	
CSD 459/559	Computer Simulation and Modeling	
CSD 467/567	Compiler Design	
CSD 470/570	Intro. to Applied Automation	
OTHER REQUIREMENTS (Tentative)		
Freshman English		6 sem. hours
Speech		3
Technical Writing		3
Literature		3
Classics, Philosophy, or Religious Studies		3
Microeconomics		3
Social Science		12
Chemistry		~~10~~ 5
Physics		~~8~~ 10
Mathematics		21
Calc./Analyt.I,II	9	
Discrete Struct.	3	
Linear Algebra	3	
Numerical Analy.	3	
Prob./Stat.	3	
Princ. of Accounting	3	
Princ. of Management		3
ELECTIVES (to bring to a total of 128 sem. hours)		6

Figure 3. Draft Curriculum Summary (Concluded)

Acknowledgement

Appreciation is expressed to the continuing participants in this process, and to the MU administrators who encouraged it from the beginning and strongly support it. Advisory Panel members: Larry Cassity, Tom Pressman, Jim Roma, Bob Shields, Mike Workman. At Marshall University: Dr. Wade Gilley, President; Dr. Alan Gould, Vice President for Academic Affairs; Dr. Steve Hanrahan, Dean of Science; CIS Faculty Dr. Hamid Chahryar, Dr. Jamil Chaudri, Dr. Akhtar Lodgher, Prof. Elias Majdalani, Dr. Wlodek Ogryczak, Dr. Dave Walker; other MU participants Dr. Dick Begley, Mr. Allen Taylor, Dr. John Wallace. Dr. Akhtar Lodgher's participation in curriculum development has been exceptionally helpful and enthusiastic. The author expresses personal appreciation to Art and Joan Weisberg for their generosity in endowing the Weisberg Chair, a recent example of their long-term interest in and support for education of the young men and women of the Tri-State region.

References

[CC91] Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force; ACM Press/IEEE Computer Society Press.

[CSAB] 1990 Annual Report for the year ending September 30, 1990; Computing Sciences Accreditation Board, Inc.

[FORD90] 1990 SEI Report on Undergraduate Software Engineering Education, CMU/SEI-90-TR-3.

APPENDIX B

Description of B.S. Degree program

COMPUTER SCIENCE AND SOFTWARE DEVELOPMENT

The Department of Computer Science and Software Development (CSD), previously the Department of Computer and Information Sciences, was transferred from the College of Business to the College of Science in 1991. The curriculum was extensively revised and updated, and now has a strong emphasis on software engineering. The new B.S. degree program emphasizes the team approach to software development and maintenance. Graduates with this orientation are very much in demand by industry and government. The program provides a solid grounding in modern computer science, including computer architecture, operating systems, algorithms, and programming languages.

The first offering of courses in the new curriculum began in Spring 1992. As new courses are offered for the first time each semester, some old courses are gradually being discontinued. All majors are strongly advised to stay in close contact with the CSD department during this transition period. Majors who began their degree programs prior to the beginning of the new curriculum should obtain handouts from the CSD department office showing which new courses to take as replacements for discontinued courses.

B.S DEGREE PROGRAM IN COMPUTER SCIENCE AND SOFTWARE DEVELOPMENT[1]

PLAN OF STUDY

FRESHMAN YEAR

Fall Semester		Spring Semester	
ENG 101 English Comp	3	ENG 102 English Composition	3
CSD 119 Intro. Computing I	4	CSD 120 Intro. Computing II	4
MTH 131 Calculus I	5	MTH 230 Calculus II	4
Social Science Elective	3	CHM 211 + Lab. Principles of Chem	5
Total	15	Total	16

SOPHOMORE YEAR

Fall Semester		Spring Semester	
Literature	3	CSD 240 Algorithm Design & Analysis	3
CSD 212 Intro. Computer Engineering	3	CSD 222 Computer Org./Assembly Lang.	3
MTH 340 Discrete Structures	3	MTH 330 Linear Algebra	3
PHY 211 + Lab Principles of Physics	5	PHY 213 + Lab Principles of Physics	5
ECN 250 Principles of Microeconomics	3	ACC 250H Principles of Accounting	3
Total	17	Total	17

1 Some College of Science general requirements do not apply to this program.

JUNIOR YEAR

Fall Semester		Spring Semester	
SPH 305 Principles of Comm	3	ENG 354 Technical writing	3
CSD 313 Intro. Systems & Software Engineering	3	CSD 333 Software Engineering	3
CSD 322 Computer Architecture	3	CSD 338 Operating Systems	3
CSD 325 Programming Languages	3	CSD Elective	3
MTH 445 Statistics	3	MTH 443 Numerical Analysis	3
MGT 320 Principles of Management	3		
Total	18	Total	15

SENIOR YEAR

Fall Semester		Spring Semester	
CSD 493 Senior Team Project	3	CSD 494 Senior Team Project	3
CSD Elective	3	CSD Elective	3
Social Science	6	Social Science	3
Classics/Philosophy/Religion	3	Free Elective	6
Total	15	Total	15

Note: the following is to appear in the section showing course offerings by department.

COMPUTER SCIENCE AND SOFTWARE DEVELOPMENT (CSD)

101 **Fundamentals of Computing. 3 hrs. I,II,S.**
Computer literacy. Introduction to fundamental concepts and skills of computing. Includes terminology, control statements, program execution, disk handling. Hands-on experience in word processing, spread sheets, databases, electronic mail. (PR: none)

119 **Introduction to Computing I. 4 hrs. I,II,S.**
Introduction to the entire system life cycle. Problem Analysis and algorithm development. Program design, coding, and testing. Introduction to the Ada language. Extensive experience in programming, including supervised lab sessions. (PR: CSD 101 or equivalent, and high school algebra)

120 **Introduction to Computing II. 4 hrs. I,II,S.**
Continuation of CSD 119, emphasizing data structures (stacks, queues, trees, graphs), and algorithms for data structure manipulation. Advanced features of Ada. Numerous programming projects, involving larger, more complex solutions. Professional ethics. (PR: CSD 119; MTH 131 co-requisite)

212 **Introduction to Computer Engineering. 3 hrs. I.**
Number system, Boolean algebra, Boolean function minimization techniques. Introduction to digital circuits and design; design and analysis of combinational and sequential circuits, asynchronous and synchronous circuits. (PR: CSD 120; PHY 211 and MTH 340 co-requisites)

222 **Computer Organization and Assembly Language Programming. 3 hrs. II.**
Introduction to PC architecture; memory architecture and management. Data representation, I/O devices. Overview of software systems: assembler, linker, debugger. (PR: CSD 212; PHY 213 co-requisite)

240 **Analysis and Design of Algorithms. 3 hrs. II.**
Review basic data structures and introduce advanced data structures; algorithm complexity analysis, identification of efficient methods. Algorithm design techniques (divide and conquer, backtracking, etc.). Intractable problems, decidability. (PR: CSD 120, MTH 340)

280-283 **Special Topics. 1-4; 1-4; 1-4;1-4 hrs. (PR: Permission of Instructor)**

313 **Introduction to Systems and Software Engineering. 3 hrs. I.**
The software development and maintenance process, software life cycle, software within a larger system; requirements analysis and specification; system engineering approaches; automated tools; requirements analysis/specification team project. (PR: CSD 240)

322 **Computer Architecture. 3 hrs. I.**
Introduction to microprocessor; design alternatives, microprogramming, bus structure, memory organization, serial and parallel port design, alternative computer architecture. (PR: CSD 222, CSD 240)

325 **Introduction to Programming Languages. 3 hrs. I.**
Comparative evaluation and use of several languages; syntax and semantics--including specification; compilation and software engineering issues; control, data, module approaches. Imperative and functional languages; concurrency, logic, object-oriented approaches. (PR: CSD 222)

333 **Software Engineering. 3 hrs. II.**
Review of requirements determination. Functional and object-oriented design; automated tools. Real-time, reliability, software reuse. Implementation, integration, testing, maintenance. Verification and validation, configuration management. Team project, large system. (PR: CSD 313)

338 **Operating Systems. 3 hrs. II.**
Process management, device and memory management, security, networking, distributed operating systems. Emphasis on the Unix operating system. Experimental projects using the C programming language. (PR: CSD 222, CSD 240)

345 **Software Development for Business/Management. 3 hrs. II, S.**
Software development and maintenance approaches for effective computing in business and management applications. Programming languages for these applications, including COBOL, fourth-generation languages. Support environments. Participation in team projects. (PR: CSD 120)

356 **Scientific/Engineering Computing and Supercomputing. 3 hrs. I, S.**
Software development and maintenance approaches for effective scientific and engineering computing and supercomputing. Languages for these applications (especially FORTRAN), including parallel approaches and vectorization. Support environments. Participation in team projects. (PR: CSD 222, CSD 240)

367 **Systems Programming. 3 hrs. II.**
Principles of systems programming; language translators, assemblers, interpreters, and compilers. Advanced operating system concepts: management of memory, I/O, files, processes. (PR: CSD 338)

409 **Software Development for Health Care. 3 hrs. II, S.**
Software development and maintenance approaches for the health care industry. Shared database approaches; instrumentation interfacing and control; inquiry/response methods and effective user interfaces. Participation in team projects. (PR: CSD 120)

419 **Decision Systems. 3 hrs. I, S.**
System/software approaches to decision support systems. On-line group decision systems, knowledge-based systems, interactive user interfacing methods, electronic conferencing and teleconferencing, statistical software, distance learning/response techniques, trends. Project participation. (PR: CSD 313 or permission)

429 **Introduction to Computer Graphics. 3 hrs. II.**
Introduction to underlying theory and techniques of computer graphics. Historical perspective. Display hardware technology, 2D raster operations, 2D and 3D geometric transformations, and 3D projection and viewing techniques. Project participation. (PR: CSD 338, MTH 330)

439 **Introduction to Artificial Intelligence. 3 hrs. I.**
Concepts and methods. Heuristic search, planning, hypothesis formation, modeling, knowledge acquisition and representation. Languages, methodologies, tools. Applications, such as automatic programming, theorem proving, machine vision, game playing, robots. Project participation. (PR: CSD 240)

442 **Communication Networks and Distributed Systems. 3 hrs. II.**
Network structures, architectures, topology. Layers, protocols, interfaces, local area networks. Coverage of current networks. Distributed processing concepts; architectural tradeoffs, distributed databases. Operating system and application software issues. Project participation. (PR: CSD 322, CSD 338)

449 **Formal Languages and Automata Theory. 3 hrs. I.**
Concepts and formalisms of formal languages and automata theory. Fundamental mathematical concepts. Grammars and corresponding automata. Deterministic parsing of programming languages. (PR: CSD 240, MTH 340)

457 **Database Systems. 3 hrs. II.**
Basic concepts, semantic models. Data models: object-oriented and relational, lesser emphasis on network and hierarchical. Query languages and normal forms. Design issues. Security and integrity issues. (PR: CSD 313, CSD 338)

459 Computer Simulation and Modeling. 3 hrs. I.

Concepts of model building and computer-based discrete simulation. Special-purpose simulation languages. Experimental design, analysis of results. Statistical aspects, random number generation. Model validation issues and methods. Project participation. (PR: CSD 313, MTH 445)

467 Compiler Design. 3 hrs. I.

Compilation of modules, expressions, and statements. Organization of a compiler including compile-time and run-time aspects; symbol tables, lexical analysis, syntax analysis, semantic analysis, optimization, object-code generation, error diagnostics. Compiler writing tools. Participation in compiler development project. (PR: CSD 325, CSD 333)

470 Introduction to Applied Automation. 3 hrs. I.

Introduction to production economics. Programmable logic control, sensors and actuators, digital and analog I/O design. Introduction to robotics and flexible manufacturing systems. (PR: CSD 322)

480-483 Special Topics. 1-4; 1-4; 1-4; 1-4 hrs. (PR: Permission of Instructor)

485-488 Independent Study. 1-4; 1-4; 1-4; 1-4 hrs. (PR: Permission of Instructor)

493 Senior Team Project Sequence, First Semester. 3 hrs. I.

With CSD 494, constitutes a year-long capstone team project, carrying out an entire system and software engineering life cycle for a project of realistic size and complexity. (PR: CSD 322, CSD 333)

494 Senior Team Project Sequence, Second Semester. 3 hrs. II.

A continuation of the project begun in CSD 493. CSD 493 and CSD 494 should be taken in consecutive semesters of the same academic year. (PR: CSD 493)

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 119

TITLE: Introduction to Computing I

CREDIT: 4 (Four)

COURSE DESCRIPTION AND PREREQUISITES:

Introduction to the entire system life cycle of software development. Problem analysis and algorithm development. Program design, coding and testing. Introduction to the Ada programming language. Extensive experience in programming including supervised lab-sessions.

Prerequisites: CSD 101 (Fundaments of Computing) or equivalent
High School Algebra

COURSE OFJECTIVES:

1. Introduce algorithmic and computational problem solving
2. Introduce software engineering methodologies for software development
3. Computational implementation of the above two objectives using a high-level language

It is important to note that the objective of this course is to teach the algorithmic and the computational problem-solving process. Prime emphasis is on the process and not the high-level language used to implement the process. Only those components of the syntax of the language will be taught which are necessary for understanding and implementing the problem-solving process.

COURSE OUTLINE:

Topics covered in this course include procedural abstraction, control structures, iteration, recursion, simple basic data types and their representation, introduction to abstract data types, etc. For each of these topics, specific problems will be studied that enhance the characteristics involved. These problems will be very carefully chosen. The solution process of these problems will be studied strictly from a software engineering perspective - explicit mention of clear specifications, top-down/bottom-up modular design, testing of design, conversion of design to code and testing of code. The Ada programming language will be used for programming exercises. Programming assignments and programming quizzes will be given.

COURSE ACTIVITIES AND EVALUATION METHODS:

There shall be three lecture hours and two hours of closed lab per week. In the closed lab, students shall get hands-on practice on programs involving the current topic being taught. In addition, there shall be an open lab where the students can come anytime and do their exercises. For every one hour of class, the student will be expected to put in three to four hours of preparation. One programming assignment and quiz will be given for every major topic (a maximum of 10 assignments). At least two exams (a midterm and a final exam) will be conducted.

TEXT:

Feldman, Michael, and Elliot Koffman, ADA: PROBLEM SOLVING AND PROGRAM DESIGN, Addison Wesley, 1991

BIBLIOGRAPHY:

Savitch, Walter, and Charles Petersen, ADA: AN INTRODUCTION TO THE ART AND SCIENCE OF PROGRAMMING, Benjamin Cummings, 1992

Volper, Dennis, and Martin Katz, INTRODUCTION TO PROGRAMMING USING ADA, Prentice Hall, 1992.

Bryan, Douglas, and Geoffrey Mendal, EXPLORING ADA, VOL 1 AND VOL 2, Prentice Hall, 1992.

Reinhold, Van Nostrand, ADA QUALITY AND STYLE: GUIDELINES FOR PROFESSIONAL PROGRAMMERS, 1989

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 120

TITLE: Introduction to Computing II

CREDIT: 4 (Four)

COURSE DESCRIPTION AND PREREQUISITES:

Continuation of CSD 119, with emphasis on data structures (stacks, queues, trees, etc), and algorithms for data structure manipulation. Advanced features of the Ada programming language will be used. Numerous programming projects involving larger, more complex solutions.

Prerequisites: CSD 119 (Introduction to Computing I)
Corequisite or completion of Math 131 (Calculus with Analytic Geometry I)

COURSE OBJECTIVES:

1. Continuation of algorithmic and computational problem solving
2. Use of intermediate level data structures and control structures
3. Continued use of software engineering methodologies for software development using the above data and control structures
4. Computational implementation of the above objectives using a high-level language

COURSE OUTLINE:

The data structures covered in this course include stacks, queues, and trees. Use of recursion and abstract data types will be emphasized. These data structures will be used for applications such as expression evaluation (Polish notation), sorting, searching, tree-traversal, etc. Simple CASE tools will be used for developing the design. There will be a continual stress on adhering to the software engineering principles: explicit mention of clear specifications, top-down/bottom-up modular design, testing of design, conversion of design to code and testing of code. An introduction to computational complexity and algorithm analysis will be given. Advanced constructs of the Ada programming language will be used. Programming assignments of larger complexity and programming quizzes will be given.

COURSE ACTIVITIES AND EVALUATION METHODS:

There shall be three lecture hours and two hours of closed lab per week. In the closed lab, the student shall get hands-on practice on programs involving the current topic being taught. In addition, there shall be an open lab where the students can come anytime and do their exercises. For every one hour of class, the student will be expected to put in three to four hours of preparation. One programming assignment and quiz will be given for every major topic (a maximum of 5 assignments). At least two exams (a midterm and a final exam) will be conducted.

TEXT:

Feldman, Michael, ADA: ADVANCED PROBLEM SOLVING AND PROGRAM DESIGN, Addison Wesley, 1992 (In print)

BIBLIOGRAPHY:

Kruse, Robert L., DATE STRUCTURES AND PROGRAM DESIGN, Prentice Hall, 1992

Tenenbaum, Aaron Y., Langsam, M. Augenstein, DATA STRUCTURES, Prentice Hall, 1992

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 313

TITLE: Introduction to Systems and Software Engineering

CREDIT: 3 (Three)

COURSE DESCRIPTION AND PREREQUISITES:

The software development and maintenance process, software life cycle, software within a larger system; requirements analysis and specification; system engineering approaches; automated tools; requirements analysis/specification team project.

Prerequisites: CSD 240 (Analysis and Design of Algorithms)

COURSE OBJECTIVES:

1. Introduce the system concept, with software as a part of a larger system
2. Emphasize the importance of a well-defined and enforced process for system and software development and maintenance, including the team approach
3. Emphasize especially requirements determination and specification
4. Introduce systems engineering methods and tools to assist requirements determination

COURSE OUTLINE:

The course will introduce systems engineering and software engineering,and their interrelationships. A brief history of the field will be studied, including a motivation for current approaches. The software life cycle will be studied, including phase interrelationships. The concepts of process and process maturity will be explored. Especial emphasis will be given to methods and tools for requirements determination. This will include CASE (computer assisted systems engineering/computer assisted software engineering) tools and their use. Object-oriented system analysis and functional-oriented system analysis will be presented. Life-cycle verification and validation and quality emphasis will be stressed, as will software reuse, human engineering, trade-off studies, metrics, cost analysis, project management, and risk analysis. Students will participate in requirements analysis and specification as members of small teams.

COURSE ACTIVITIES AND EVALUATION METHODS:

This course will consist of three lecture hours per week, and extensive out-of-class team project participation. As team projects are conducted, formal reviews will be conducted during class periods, during which teams will review their activities and status, including specifications. Other students will critique the material presented, and submit suggested changes. A mid-term exam. and a final exam. will be conducted.

TEXT: (The following textbook is a likely choice at this time--and in any case is representative of the material to be covered.)

Sage, Andrew P. and James D. Palmer, SOFTWARE SYSTEMS ENGINEERING, John Wiley and Sons, 1990.

BIBLIOGRAPHY:

Davis, Alan M., SOFTWARE REQUIREMENTS: ANALYSIS AND SPECIFICATION, Prentice Hall, 1990.

Humphrey, Watts S., MANAGING THE SOFTWARE PROCESS, Addison-Wesley, 1989.

Pressman, Roger S., SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, 3rd ed., 1992.

Selected current journal articles.

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 333

TITLE: Software Engineering

CREDIT: 3 (Three)

COURSE DESCRIPTION AND PREREQUISITES:

Review of requirements determination. Functional and object-oriented design; automated tools. Real-time, reliability, software reuse. Implementation, integration, testing, maintenance. Verification and validation, configuration management. Team project, large system.

Prerequisites: CSD 313 (Introduction to Systems and Software Engineering)

COURSE OBJECTIVES:

1. Continue to emphasize the importance of well-defined, repeatable approaches to software engineering
2. Study functional and object-oriented approaches to software design
3. Emphasize user interface issues
4. Emphasize effective implementation with Ada
5. Study effective methods for integration and testing
6. Provide further experience in the team approach to software development

COURSE OUTLINE:

Review requirements specifications, and introduce both functional and object-oriented methods for design. Emphasize software reuse as a means to improve productivity and quality. Emphasize effective user interface techniques. Conduct team activities, including the use of CASE (computer assisted software engineering) tools. Present methods of unit testing, integration, and system testing. Discuss issues of real time, safety, and configuration management. Further emphasis on software management, and software maintenance.

COURSE ACTIVITIES AND EVALUATION METHODS:

This course will consist of three lecture hours per week, and extensive out-of-class team project participation. As team projects are conducted, formal reviews will be conducted during class periods, during which teams will review their activities and status, including designs and code. Other students will critique the material presented, and submit suggested changes. A mid-term exam. and a final exam. will be conducted.

TEXT: (The following textbook is a likely choice at this time--and in any case is representative of the material to be covered.)

Sommerville, Ian, SOFTWARE ENGINEERING, 4th ed., 1992.

BIBLIOGRAPHY:

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, 1991.

Booch, Grady, SOFTWARE ENGINEERING WITH ADA, 3rd ed., Benjamin/ Cummings, 1992.

Selected current journal articles.

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 493

TITLE: Senior Team Project Sequence, First Semester

CREDIT: 3 (Three)

COURSE DESCRIPTION AND PREREQUISITES:

With CSD 494, constitutes a year-long capstone team project, carrying out an entire system and software engineering life cycle for a project of realistic size and complexity.

Prerequisites: CSD 322 (Computer Architecture), CSD 333 (Software Engineering)

COURSE OBJECTIVES:

The objectives for CSD 493 and CSD 494 are as follows:

1. Provide a realistic experience for seniors in large-scale software development and maintenance.
2. Have the students work on a "real" problem, for a "real" customer, thus help prepare them for effective performance in the workplace
3. Give the students the opportunity to apply much of what they have studied in their earlier courses

COURSE OUTLINE:

CSD 493 and CSD 494 together will provide the opportunity for the students to apply the methods and automated tools they have previously studied to a large-scale, realistic problem, and to interact with a "customer" in a realistic way. The entire life cycle of software development will be carried out, and the students will have some experience in performing maintenance on existing software. Some experience will be gained in software reuse. Production of various documents, and preparation for and participation in formal reviews, will be part of the experience. The instructor will present lectures on advanced software engineering material for which there was not time in earlier courses, paced to be useful in the team project. This will include formal methods, detailed configuration management procedures, IEEE and U.S. Department of Defense standards for software engineering activities and products, further study of cost modeling and metrics, additional emphasis on system testing, and emphasis on the Software Engineering Institute's process evaluation and improvement methods.

COURSE ACTIVITIES AND EVALUATION METHODS:

Each of CSD 493 and CSD 494 will consist of approximately 1.5 lecture hours per week, with approximately 1.5 class hours per week being devoted to formal reviews and walkthroughs, and interaction with the instructor on the work underway. Extensive out-of-class team project work will be necessary. The project work in CSD 493 and CSD 494 will be a continuum, with the ending point of CSD 493 corresponding to a major life cycle milestone (e.g., preliminary design complete). A mid-term exam. and a final exam. will be conducted in each of CSD 493 and CSD 494.

TEXT:

The textbooks for CSD 313 and CSD 333 will be used for reference and additional lecture material in CSD 493 and CSD 494.

BIBLIOGRAPHY:

The reference books for CSD 313 and CSD 333 are applicable to CSD 493 and CSD 494.

Additionally, IEEE standards documents and DOD-STD-2167A (Department of Defense Software Engineering Standards) will be used for reference.

Also, one or more books on configuration management will be used, as will current material from the Software Engineering Institute, and selected current journal articles.

COURSE CHANGE/NEW COURSE INFORMATION

DEPARTMENT AND COURSE NO: Computer Science and Software Development
CSD 494

TITLE: Senior Team Project Sequence, Second Semester

CREDIT: 3 (Three)

COURSE DESCRIPTION AND PREREQUISITES:

A continuation of the project begun in CSD 493. CSD 493 and CSD 494 should be taken in consecutive semesters of the same academic year.

Prerequisite: CSD 493 (Senior Team Project Sequence, First Semester)

COURSE OBJECTIVES:

The objectives for CSD 493 and CSD 494 are as follows:

1. Provide a realistic experience for seniors in large-scale software development and maintenance.
2. Have the students work on a "real" problem, for a "real" customer, thus help prepare them for effective performance in the workplace
3. Give the students the opportunity to apply much of what they have studied in their earlier courses

COURSE OUTLINE:

CSD 493 and CSD 494 together will provide the opportunity for the students to apply the methods and automated tools they have previously studied to a large-scale, realistic problem, and to interact with a "customer" in a realistic way. The entire life cycle of software development will be carried out, and the students will have some experience in performing maintenance on existing software. Some experience will be gained in software reuse. Production of various documents, and preparation for and participation in formal reviews, will be part of the experience. The instructor will present lectures on advanced software engineering material for which there was not time in earlier courses, paced to be useful in the team project. This will include formal methods, detailed configuration management procedures, IEEE and U.S. Department of Defense standards for software engineering activities and products, further study of cost modeling and metrics, additional emphasis on system testing, and emphasis on the Software Engineering Institute's process evaluation and improvement methods.

COURSE ACTIVITIES AND EVALUATION METHODS:

Each of CSD 493 and CSD 494 will consist of approximately 1.5 lecture hours per week, with approximately 1.5 class hours per week being devoted to formal reviews and walkthroughs, and interaction with the instructor on the work underway. Extensive out-of-class team project work will be necessary. The project work in CSD 493 and CSD 494 will be a continuum, with the ending point of CSD 493 corresponding to a major life cycle milestone (e.g., preliminary design complete). A mid-term exam. and a final exam. will be conducted in each of CSD 493 and CSD 494.

TEXT:

The textbooks for CSD 313 and CSD 333 will be used for reference and additional lecture material in CSD 493 and CSD 494.

BIBLIOGRAPHY:

The reference books for CSD 313 and CSD 333 are applicable to CSD 493 and CSD 494.

Additionally, IEEE standards documents and DoD-STD-2167A (Department of Defense Software Engineering Standards) will be used for reference.

Also, one or more books on configuration management will be used, as will current material from the Software Engineering Institute, and selected current journal articles.

APPENDIX C

"Teaching at the Senior Level"

TEACHING ADA AT THE SENIOR LEVEL[1]

Akhtar Lodgher, Ph.D
Department of Computer and Information Sciences
Marshall University
Huntington, WV - 25755
Email: CIS005@MARSHALL.WVNET.EDU
Phone: (304)-696-2695, Fax: (304)-696-4646

INTRODUCTION

The requirements of teaching Ada at the senior level are different than teaching Ada at the introductory level. At the introductory level, the emphasis is more on the problem solving skills, and hence only those parts of the language are used which are necessary for building such skills. At the senior level however, the students are well trained in problem solving skills. More emphasis must thus be placed on learning the variations in syntax. Also, since Ada has so many additional features (compared to other imperative languages) such as packages, generics, tasking, low-level representation, and exceptions, a thorough base of each of these features must also be taught. In this paper, the structure and the contents of the various aspects of such a course, designed and taught by the author, are discussed.

APPROACH

Since the amount of material to be covered is large, a "hands-on" approach to teaching is used. This means that the instructor have the capability of showing "real" Ada programs, modifying them and executing them in the class. This is accomplished with the use of an overhead video projector. A lot of writing on the black board is avoided by using class notes in electronic form. Class notes are made using a word processor, and customized for every class (unlike overhead transparencies). The beauty of this approach is that example programs and notes from various sources could be scanned electronically and mixed according to the instructor's style. The notes are made available to the students before class (public files on a mainframe system). This enables them to get printouts, and spend time listening in class rather than writing notes *and* listening.

There is not a single specific text book that covers all the above topics in the detail required. There are the introductory books (such as Feldman and Koffman's *Ada: Problem solving and program design*, and Savitch and Petersen's *An introduction to the art and science of programming*) which have many example programs and include a program disk. This makes them an easy source to draw sample programs from. However, the advanced topics are not covered, and one has to rely on books such as Booch's *Software Engineering with Ada* or Gehani's *Ada: An advanced introduction*. Such books generally do not provide program disks. If they do, the example programs which explain a particular concept of the language syntax or semantics are program segments, not complete programs. This makes it especially hard for the "hands-on" approach of modifying and executing programs, forcing the instructor to spend more

---

[1] Presented at the Sixth Southeastern Small College Computing Conference, Nov 6-7, 1992, Jefferson City, TN.

time typing or scanning the programs.

One of the biggest advantage of the "hands-on" approach is that motivation of the students is kept high, and they are forced to be alert in class. Questions such as "What will happen if such and such" are answered by very quickly implementing the question in a program, executing the program, and displaying the results. This increases retention. The use of electronic notes gives the capability of switching between the notes and programs. Once a particular syntax or semantic concept is explained, an example program emphasizing that concept can be presented and executed.

However, the disadvantage of using sample programs from different sources is that each author follows an individual style of programming. The style of some of the programs could be considered deviant from the accepted style. Since most students have the attitude of "do what you see", presenting such programs to them would cultivate bad style. Thus, some effort must be spent in converting such deviant programs into an acceptable form before presentation.

COURSE CONTENTS AND WEEKLY TOPICS

The contents are discussed for a 15 week semester. The course is divided into two parts: the introductory part and the advanced part.

The introductory part is covered in the first seven weeks and includes the following topics (in order): introduction to the language, data types (subtypes and attributes), input/output packages, procedures and functions, control structures (selection and looping), introduction to exceptions, arrays, records (variant) and access types. The students are seniors and are familiar with each of these concepts in some form or another. Thus more emphasis is placed on the syntax of these structures. However, the concepts of subtypes, attributes, and exceptions are completely new and thus required more explanation time. Each of the other features have a difference (an enhancement when compared to Pascal) in implementation, such as slices in arrays and variations in the variant record structure. Example programs that use these newer capabilities are used for explanation.

The advanced part is covered in the next eight weeks and the following topics (in order) are covered: packages, generics, tasking and low-level representation. About two weeks are spent on each topic. The introduction of packages introduces the concept of process modularity over a set of subprograms and the concept of separate compilation units. The propagation of exceptions through different packages is introduced at this point. Example programs that explain each of these concepts are essential. Generics introduce the concept of passing data types as parameters. Tasking introduces the concept of parallel processing. Due to the complex nature of the concepts of tasking it is discussed in a different section. Finally, low-level (machine) representation introduces concepts of accessing memory locations through representation attributes, representation clauses and pragmas. Proper implementation of each of these concepts is emphasized by using example programs.

TASKING

Tasking concepts are divided into simple, simple rendezevous, transfer of data on rendezevous, selective wait, conditional entry and task attributes.

Simple tasking introduces concepts of subprograms running in parallel, without any interaction. It brings out the behaviour of the software or hardware platform (which is generally sequential) on parallel programs. Time-slicing is emphasized by varying the execution time of each subprogram. The syntax involved is explained.

Simple rendezevous introduces the concept of one task waiting for another for a "rendezevous". Programs which show examples of different situations in which a rendezevous can occur, or not occur (tasking error) are used. The effect of loops and the use of counters and delays to track the sequence of calls, is essential.

Transfer of data is possible at rendezevous and the use of the three parameter modes is explained with the use of appropriate programs. It is also possible that an acceptor task have more than one (multiple) accept calls.

Selective wait is the concept of selecting a particular task from a pool of calling tasks. Variations include guarded selective wait, tasks with loops, use of the ELSE, DELAY, TERMINATE and ABORT statements. Various combinations of each of these variations is essential for a proper understanding. Also, some combinations do not work and the reasons for their behaviour is also explained by using example programs.

Conditional entry is the concept of the caller task checking to see if the acceptor task is busy. A variation here is the use of the DELAY statement.

Task attributes reveal the status of the different task parameters. Examples of attributes are the COUNT attribute, the CALLABLE attribute, the TERMINATED attribute, and the different PRAGMA's. Also the use of variables of a task type is explained.

STYLE

Concepts of programming style are strictly adhered to for every program submission. The style principle follwed are from the book *Ada Quality and Style* by the Software Productivity Council. A summary of the pertinent style principles is created and made available to the students. Many examples of programs adhering to individual and collective principles are made available. Where possible, specific examples of dilemmas created by not using the principles (or abusing the principles) ware also shown. Specific instructions about creating header documentation and incode documentation is also given.

ASSIGNMENTS AND QUIZZES

Assignments are longer programming projects done over a period of a week or two weeks. Quizzes are smaller programming projects done over the weekend. The purpose of an assignment is to test the implementation of a concept on a large project, whereas a quiz tests the understanding of subconcepts within a concept or a smaller concept. There are 10 programming assignments and 5 quizzes given throughout the course. The objective of each of the assignment is:

A1: Simple program to illustrate control structures (conditional and iteration)
A2: Use of subprograms and parameter modes
A3: Use of arrays and array attributes
A4: Use of the variant record structure
A5: Use of access types
A6: Use of packages and introduction to compilation units
A7: Use of generics
A8: Use of tasking (simple)

A9: Use of tasking (complex)
A10: Use of low-level representation

The objective of each of the quiz is:
Q1: Use of the looping construct
Q2: Use of procedure parameter modes
Q3: Use of string data type
Q4: Use of multi-dimensional arrays
Q5: Use of access types

Quizzes for the advanced topics are not given because of the complexity involved. The length (lines) of the assignments is about 1500-2000 lines (including documentation) and the quizzes is 100-200 lines.

In the spirit of the "hands-on" approach, the executable versions of the solutions of the assignments and quizzes is always made available before the due date. This enables the student to get a "feel" of the behaviour of the program. Also it motivates them to try to create input data which would crash the solution. After the due date, the source code of the solution is made available to the student. Each concept used, and the style principles used, are explained in detail. The students are encouraged to get printouts of the solution and study them. If there is a violation of the use of an explained concept (semantic or style) in any subsequent submission, a heavy penalty is levied.

CONCLUSION

Teaching Ada at the senior level poses special problems. The use of the "hands-on" technique is successfully used to cover the depth and breadth of the features of the language. The various programs being used in the process will be made available on a disk at the time of the presentation

REFERENCES

Grady Booch, *Software Engineering with Ada*, Benjamin Cummings, 1987.
Narain Gehani, *Ada: An Advanced Introduction*, Prentice Hall, 1989.
Michael Feldman, *Ada: Problem Solving and Program Design*, Addison-Wesley, 1992.
Geoffrey Mendal and Douglas Bryan, *Exploring Ada, Volume 1 & 2*, Prentice Hall, 1992.
Walter Savitch and Charles Petersen, *Ada: An Introduction to the Art and Science of Programming*, Benjamin-Cummings, 1992.
Ada Quality and Style, The Software Productivity Consortium, Van Nostrand Reinhold, 1989.

APPENDIX D

Syllabus of CS1 taught in Ada

Course : CIS 139, Computer Programming I, Section 101, Marshall University
Instructor : Dr. Akhtar Lodgher
Term : Fall 1992
Class : Mon, Wed, Fri 1:00 - 1:50pm, Corbly Hall 335
Lab : Wed 3:00 - 4:50, Corbly Hall 436
Office hours : Mon, Fri - 9:00 - 11:00 am, Wed 2:00 - 3:00 pm
Office : Corbly Hall 331B, Phone: 696-2695
Mail box & Secretary: Corbly Hall 310

SYLLABUS

GOALS/OBJECTIVES OF THE COURSE:

1. Introduce algorithmic and computational problem solving
2. Introduce software engineering methodologies for software development
3. Computational implementation of the above two objectives using a high-level language

It is important to note that the objective of this course is to teach the algorithmic and the computational problem-solving process. Prime emphasis is on the process and not the high-level language used to implement the process. Only those components of the syntax of the language will be taught which are necessary for understanding and implementing the problem-solving process. The Ada programming language will be used to teach the problem solving process.

PREREQUISITES:

Computer Science: The introductory computer science course CIS 109. Students who have not had CIS 109 will not be allowed to take this course. For exceptional cases, a waiver letter from the chairman of the computer science department will be required.

Math requirement: Students must be taking in the current semester or completed successfully MTH 130 (College Algebra). Students not meeting this prerequisite will not be allowed to take this course unless a waiver letter from the chairman of the computer science department is obtained.

METHODOLOGY:

This course is designed to be a four credit course. There shall be three lecture hours per week and two hours of closed lab. In addition, there shall be an open lab where the students can come anytime and do their exercises.

Topics covered in this course include procedural abstraction, control structures, iteration, recursion, simple basic data types and their representation, introduction to abstract data types, etc.

For each of these topics, specific problems will be studied that enhance the characteristics involved. These problems will be very carefully chosen. The solution process of these problems will be studied strictly from a software engineering perspective - explicit mention of clear specifications, top-down/bottom-up modular design, testing of design, conversion of design to code and testing of code. A high-level language will be used to implement the code.

In order that the students graduating from this class have a consistent style, a standard style manual will be used. It will be mandatory for the students to follow this style manual for program style, coding and documenting.

APPROACH:

The following concepts shall be covered, in the sequence mentioned during the course. A weekly schedule of the topics is attached.

1. *An introduction to the process of problem solving using algorithms*
2. *An introduction to computational problem solving*
3. *An introduction to imperative-language being used*
4. *The use of modular design in the problem solving process*
5. *Problems involving the use of decisions*
6. *Problems involving the use of repetitions*
7. *Introduction to error handling*
8. *Introduction to the concept of data types*
9. *Problems involving the use of scalar data types*
10. *Problems involving the use of array data type*
11. *Problems involving the use of record data type*
12. *Problems involving explicit specifications and design*
13. *Problems involving the use of Abstract data types*
14. *Problems involving the use of generics*
15. *Problems involving the use of dynamic data types*
16. *Problems involving the use of recursion*

TEXT BOOK AND MANUALS:

Required:

1. **Ada Problem Solving and Program Design**, by Michael Feldman and Elliot Koffman, Addison Wesley, Reading, MA, 1991, ISBN 0-201-50006-X (without disk). Available in college bookstore

2. **Introduction to VAX VMS Manual**, by Akhtar Lodgher, available at Kinkos

3. **Ada Language Design, Style and Documentation Manual**, by Akhtar Lodgher, available at Kinkos, packet 17

4. **CIS 139, Computer Programming I, Lab Manual**, by Akhtar Lodgher, available at Kinkos

MATERIAL COVERED IN CLASS:

Most of the topics covered will be in the text. Additional material may also be covered in the class. You are responsible for all the material presented in class and the lab in the form of lecture, notes and handouts. In case you are not present for a class, it is your responsibility to contact the instructor and receive information about the material presented in that class.

ASSIGNMENTS, LABS AND EXAMS:

There will be 10 programming assignments to be handed in during this course. Each of these will be a programming project.

Every lab session will require you to do a few exercises in the lab.

There shall be two exams:
MID-TERM EXAM : Oct 14, 1992, Wednesday, 3:00 - 4:50 pm
FINAL EXAM : Dec 10, 1992, Thursday, 1:00 - 3:00 pm

All assignments, labs, and exams are required parts of this course and must be satisfactorily completed to pass this course. Concepts learned in one assignment or lab are used in other assignments or labs. Thus a fail grade on any of the assignments or labs may severely affect later performance. The student must have a passing performance on the aggregate of the assignments, the aggregate of the labs and the aggregate of the two exams. A failing grade on either of these aggregates **WILL** result in a fail grade in the course. Late or make-up exams, assignments or labs **WILL NOT BE GIVEN**. Assignments and labs submitted after the due date and time will not be accepted. Failure to attend either of the two exams **WILL** result in a fail grade

GRADE DETERMINATION:

20 percent for labs
40 percent for the programming assignments
20 percent for the midterm exam
20 percent for the final exam

The instructor reserves the right to change these values, depending upon class performance and/or extenuating circumstances.

Letter grades will be used in determining final grades.

COMPILERS:

For all the programming assignments, quizzes and exams, you have to use the Ada compiler available on the Vax. If you have your own Ada compilers for PC's, then you must upload your programs onto the Vax. Computer accounts for the Vax, which will be valid for the current semester only, will be given to each student.

ASSIGNMENT AND LABS SUBMISSION:

- The design (the DFD and Structure chart) of the assignment is due every Friday before 4:30 pm and must be put in my mail box. If the design is not handed in, the code of the assignment will not be accepted.

- The source code of all assignment programs must be submitted to account (CIS0006@muvms2) before 1:00 pm every Monday. The files must be named exactly as required.

- The compiled listing of the assignment, along with the output must be handed in class at the beginning of class.

- The lab exercises must be completed and submitted to the above account before 4:30pm Fridays. All listings and output must be handed in before 4:30 pm Fridays.

- Delay in submission, even by one second, will be considered late, and a grade for that assignment or lab will not be given.

ASSIGNMENT GRADING:

Each assignment will be worth 100 points, and will be graded as follows:

50 points	For the design (DFD, Structure chart)
25 points	For programming style (if the code is doing what it is supposed to do)
25 points	For a correctly working program

COMMUNICATION:

The Mail facility of the Vax computer will be used to make any general announcements, last minute changes, etc. It is mandatory that you monitor your mail messages at least twice every day, once in the morning and once in the evening.

HONOR CODE:

You are expected to abide by the honor code at all times. During in-class exams, no collaboration or discussion or use of reference aids is allowed, unless otherwise stated at the time the exam is given. All labs and assignments are to be individual efforts, and no collaboration with members of the class or outside will be tolerated, unless otherwise stated by the instructor. If anyone sees a violation of the Honor Code, please report it to the instructor immediately. Any violation of the Honor Code will be reported to the chairman, the dean and the Honor Committee of the University.

COURSE WITHDRAWAL POLICY:

A student who does not attend the class in the first week of classes will be dropped automatically administratively.

A student who withdraws on or before Friday, Oct 16, 1992 will get an unconditional grade of "W".

A student who wants to withdraw between Monday Oct 19, 1992 and Friday, Nov 13, 1992 will be evaluated on his or her performance based on the above grading weights. If the student has passing grade, he or she will get a "WP", else a "WF". **This policy will be very strictly enforced.**

After Monday, Nov 16, 1992 individual course withdrawls will not be permitted under any circumstances.

APPENDIX E

"Using Ada for a Team Based Software Engineering Approach to CS1"

Accepted for presentation and publication in the proceedings of the 11th Annual National Conference on Ada Technology. March 15 - 18, 1993, Williamburg, VA.

USING ADA FOR A TEAM BASED SOFTWARE ENGINEERING APPROACH TO CS1

AKHTAR LODGHER & JAMES HOOPER

Department of Computer Science and Software Development
Marshall University
Huntington, WV - 25755
Phone: (304)-696-2695 Fax: (304)-696-4646
Internet: CIS005@marshall.wvnet.edu

In the past year, the Computer Science department at Marshall University has revised the Bachelor's degree program, and given a very strong emphasis to software engineering throughout the entire curriculum.[1] The department decided to use Ada as the standard programming language for the first few courses. In later courses, exposure to other languages such as C and C++ is also given. The program has two capstone courses, taken in the last two semesters, where a major team project is designed and implemented. Hence the need for emphasizing software engineering principles, as well as getting students used to programming in teams from the very first computer science course was strongly felt. In this paper, the author presents the syllabus and a method of executing the syllabus of the CS1 course satisfying the above needs. Software engineering principles are introduced early on, and after an initial boot-strapping period, the programming projects are done in teams. The Ada programming language is used.

Introduction

CS1 is taught as a 4 semester-hour course in a 16 week semester. The students attend three hours of lecture a week and two hours of closed lab. Concepts introduced in the class are reinforced in a closed lab setting. An open lab is also available for students to complete their lab exercises and programming projects. The Ada compiler on a VAX/VMS system is used. Students are allowed to use PC based compiler environments for program development. However all work is graded only on the VAX.

Syllabus

The objective of this course is to develop problem analysis and algorithm development skills. Topics covered in this course include introduction to the entire life cycle of software development, introduction to the use of modular design in the problem solving process, procedural abstraction, decision structures, iteration structures, basic data types, array and record structures, abstract data types, use of generic code, and introduction to dynamic structures. Problems that enhance the characteristics of each concept/structure are used. The problem solving process is emphasized over language implementation. An example of this principle for illustrating the looping process is: "Let us study this problem (which requires a loop construct) and develop an algorithm for its solution" rather than "These are the looping constructs available in this language. Let us see the kinds of problems that can be solved using these constructs".

The solution process of these problems is studied strictly from a software engineering perspective – conducting a requirements specification and analysis, performing a modular top-down design, development of module specifications, adherence of code to design. From the very first class, the students are told to perceive themselves as software engineers and designers, not programmers.

A team approach is used for programming assignments - two students per team. One person of the team does the design and the other person develops the code based on the design. For the next assignment, the roles are switched. This approach forces the designer to conduct a proper analysis and design. The "coder" has to follow the design, making only necessary changes, if required.

Approach

The "hands on" approach

The amount of material covered in the class is quite large. To ensure that enough exposure is given to each topic, a "hands on" approach to instruction is used. Programs which exhibit the characteristics of a particular concept or structure are made available to the students. These programs are displayed, explained, and executed in the classroom, on a computer using the overhead video projector. Unlike the traditional "chalk-and-talk" approach, this approach not only shows the syntax of the structure, but also shows how the structure is used in the context of a larger solution process. Minor variations and nuances of the structure are also explained.

Another important advantage of this approach is to show the possible incorrect ways of using the structure. When a student starts using a new structure, the chances of him or her using it incorrectly are high. By using incorrectly formed structures (both syntacticaly and semantically incorrect), the error messages generated are shown. The mechanism of using the error messages to trace the error in the structure can be demonstrated.

Class notes in electronic form, as well as all classroom demonstration programs are made available to the students (on a mainframe) before the class. The students are encouraged to bring a printout of the notes and the programs to the classroom. This allows them to spend time listening and participating in the classroom discussion and not be bogged down by the task of taking notes.

Other advantages of the hands on approach include increase in student participation (answering "what-if" questions), increase in understandability and increase in programming confidence. However, this approach places a tremendous burden on the instructor. The development and preparation of pedagogical examples takes a lot of time. Instruction materials associated with text books are not available in electronic form. Such material must either be scanned or typed in and fine-tuned depending upon the audience.

Course contents and weekly topics

Table 1 shows the classroom topics, the assignment and lab topics on a weekly basis. It is assumed that the student has little or no knowledge of the operating system. However, it has been found from past experience that students who have had an introductory course on computers in high school are more patient and quicker in learning the new operating system.

The first two weeks introduce the entire system life cycle of software development. A top-down analysis and design methodology is discussed next. The process of converting a problem statement to requirements specifications, analysis and design for simple problems is explained. Currently the analysis is done using data flow diagrams (DFD's) and the design using structure charts. A design manual[2] which explains this process in a step by step fashion is made available. The issue of using object-oriented design is under consideration.

The use of functions, procedures and packages is introduced early on, in the context of modular design. All the intricacies of procedures and packages are not covered at this point. Only the concept and their usage in simple contexts are covered. The branching and looping constructs are covered next.

Exception handling and the more detailed use of functions and procedures are then explained. The concept of abstract data types is introduced. The array and record structures are covered next. Examples of the use of array and records to implement abstract data types are explained. It is at this point that a more detailed explanation of packages, passing execptions, etc., are discussed.

The concept of code abstraction is explained using the generic structure on a sorting example. Finally, an introduction to dynamic data structures is given. The creation of dynamic variables and their use in creating linked lists and traversing linked lists is covered. It should be noted that the concept of recursion is not introduced in this course.

Assignments and laboratory exercises

A total of eight programming assignments are given. Of these, the first three are of an introductory nature and are done on an individual basis. The latter five assignments are done in teams. The first assignment, which is not given until the fourth week of classes, is of the nature of a "hello world" program. The second assignment involves some output formatting and the third assignment is based on the use of selection statements.

Each assignment requires the preparation of a design document. This document consists of: (a) the problem statement (b) requirements specifications (c) analysis - the data flow diagrams (d) design - structure chart showing the modules (e) module design specifications indicating the input, output and processing of each module. The design document is mandatory and must be submitted before starting the code. The code is based on the design, and the close relationship between the structure chart and the actual code is emphasized. The simple nature of the first three assignments helps in ironing out the details and links between design and code.

Beginning with the fourth assignment, the size and the complexity increases. At this point the class is divided in teams of size two. The members are chosen using a draw. One person is responsible for the design document and the other is responsible for the code. The roles for the next assignment are then switched. The following policy for grading team based assignments is set:

1. Essentially, each person gets the grade for the work done by him/her. Each assignment is worth 100 points.
2. If the design document is perfect, then the designer gets 100 points.
3. If the code follows the design and is perfect, the person in charge of code gets 100 points.
4. If the design is correct, and the code is incorrect, points are taken off from the coder.
5. If there are flaws in the design document, the designer loses points.
6. If there are flaws in the design, and the coder codes it following the design (resulting in badly designed code, though correct) the coder is penalized a little for not attempting to fix the design
7. If there are flaws in the design, and the coder fixes the design the coder gets additional bonus points for the extra effort.
8. The coder shall EXPLICITLY point out the changes in design.
9. The coder shall not unnecessarily change the design. If this is done, points are taken off from the coder.
10. If design is submitted but code is not submitted or does not work then the designer gets the points for his/her design, the coder does not get any points for his/her code. The coder is classified as a "BAD PERSON".
11. If design is not submitted, or is so bad that it is not worth following then the designer does not get any points for the design. The designer is classified as a "BAD PERSON". The coder then has to do both the design and the code. If the coder does just the code, he/she gets 100 points for the code. If the coder also does a good job on design, then bonus points are given to the coder for the design.
12. If a member is classified as a bad person twice, then on the first chance available, that member is dropped from the team and the good person combined with another good person.
13. If a team member drops the course then the left over member will be combined with an available member. If such a member is not

available, then the remaining member must do both the design and the code.

The team policy ensures that the designer conducts a proper analysis and design and the coder understands and follows the design. Initially, some friction between the team members was observed, but after a while, the members were able to work around their schedules. For larger assignments, parts of the design and code are given by the instructor.

Some amount of class time and lab time is devoted to discussing the assignments. The executable solution of each assignment is made available before the due date. This enables the students to understand the input and output format. The students can also test the performance of their program on certain input data and compare it against the instructors' solution. After the assignment is due, the solution of the assignment is shown to the student, and the design and code are discussed.

The laboratory exercises are conducted in a closed laboratory environment. A lab manual[3] which has exercises based on the text and class material is made available. The objective of each of the exercises is explained first and then the students are allowed to complete the work. The first few lab exercises familiarize the student with the operating system and the Ada compilation environment. Most of the other lab exercises consist of incomplete or incorrect programs which the students have to complete, correct or enhance.

Conclusions

The CS1 course was taught by the author, using the above syllabus, for the first time in Fall 1992. The author has taught the course many times in Pascal and he observed that the software engineering/Ada combination led to better solution designers. Enforcing the completion of design before starting code helped the students understand the solution process much better. They were able to find more flaws in the design. The modular design and development helped them to quickly find problem areas and fix them. The closed lab environment definitely helped the students in reinforcing the concepts learned in the classroom. The number of assignments may be reduced by one or two by combining concepts. The CS2 course based on this approach of CS1 is currently under preparation.

References

1. Hooper, James, *"Planning for Software Engineering Education Within a Computer Science Framework At Marshall University"*, Sixth Software Engineering Institute Conference on Software Engineering Education, Oct 5-7, 1992, San Diego.

2. Lodgher, A., *"Ada Language Design, Style and Documentation Manual"*, Department of CSD, Marshall University.

3. Lodgher, A., *"CS1 - Computer Programming I Lab Manual"*, Department of CSD, Marshall University.

Akhtar Lodgher (Ph.D 1990, George Mason University) is an Assistant Professor in the Department of Computer Science and Software Development (CSD) at Marshall University since Sept 1990. His teaching and research interests are in the fields of software engineering, data structures, algorithms and object oriented programming.

James Hooper (Ph.D 1979, University of Alabama, Birmingham) is a visting Professor, from the University of Alabama, Huntsville, occupying the Arthur and Joan Weisberg Chair in Software Engineering at Marshall University since Fall 1991. His teaching and research interests include software engineering (especially software reuse and the software process), programming languages and discrete event simulation.

Class Topics	Assignments Designs due beginning of week Programs due middle of week	Laboratory Laboratory work is due end of week
WEEK 1: - Introduction to problem solving - Software Engg Life cycle		LAB 1: - Introduction to the operating system - File structure, network characteristics
WEEK 2: - Introduction to computational problem solving - Overview of high-level languages, syntax, semantics		LAB 2: - Operating system continued, Ada environment - Use of editor, batch files
WEEK 3: - Top down design, drawing DFD's, structure charts		LAB 3: - Simple Ada programs, DFD's, input, output
WEEK 4: - Introduction to imperative languages - Scalar data types, I/O formatting - Constants, variables, types	- Handout A1 (simple program)	LAB 4: - Run simple Ada programs, show execution of A1
WEEK 5: - Introduction to the use of functions, procedures and packages	- Design of A1 is due, program of A1 is due - Handout A2 (I/O formatting)	LAB 5: - Discuss A1, show execution of A2 - Building programs, debugging
WEEK 6: - Boolean expressions, concepts of branching - IF-THEN-ELSE statement, CASE statement	- Design of A2 is due, program of A2 is due - Handout A3 (Branching)	LAB 6: - Discuss A2, show execution of A3 - Use of IF-THEN-ELSE and CASE statements
WEEK 7: - Use of repetetion, LOOP - EXIT - FOR Loop, WHILE Loop, Nested Loops	- Design of A3 is due, program of A3 is due - Handout Team A4 (Procedures and loops)	LAB 7: - Discuss A3, show execution of A4 - Use of Loops
WEEK 8: - Enumerated types, attributes of scalar, float types - Parameters to procedures and functions - Error handling (exceptions)	- Discuss A4 - Handout Team A5 (Exceptions, packages)	LAB 8: - Discuss A4 - Attributes of scalar and float types - Procedures and functions

Table 1: Weekly schedule of classroom topics, assignments and laboratory exercises

Class Topics	Assignments	Laboratory
<u>WEEK 9:</u> - Visibility issues - Abstract data types, packages, file I/O	- Design of A4 is due, program of A4 is due	<u>LAB 9:</u> - Discuss A4, show execution of A5 - Exceptions, packages
<u>WEEK 10:</u> - Use of arrays, strings multi-dimensional arrays	- Design of A5 is due, program of A5 is due - Handout Team A6 (Arrays, records, file I/O)	<u>LAB 10:</u> - Discuss A5, show A6 - Arrays, multi-dimensional arrays
<u>WEEK 11:</u> - Use of records, hierarchical records - Variant records	- Discuss A6	<u>LAB 11:</u> - Discuss A6 - Records, hierarchical and variant records
<u>WEEK 12:</u> - More on packages, passing exceptions - Generics	- Design of A6 is due, program A6 is due - Handout Team A7 (Generics, exceptions)	<u>LAB 12:</u> - Discuss A6, show execution of A7 - Packages, generics
<u>WEEK 13:</u> - Generics continued - Intro to access types	- Discuss A7 - Handout Team A8 (Access types)	<u>LAB 13:</u> - Discuss A7 - Generics
<u>WEEK 14:</u> - Access types continued	- Design of A7 is due, program A7 is due	<u>LAB 14:</u> - Discuss A7, show execution of A8 - Use of dynamic variables (Access types)
<u>WEEK 15:</u> - Review	- Design of A8 is due, program A8 is due	<u>LAB 15:</u> - Discuss A8 - Access types continued

Table 1: ..continued

APPENDIX F

Syllabus and team projects for standalone Software Engineering Course

COURSE SUMMARY

CSD 479/579 SOFTWARE ENGINEERING

Spring 1993

Monday 6:30 - 9:00 P.M., Corbly Hall Rm. 354

INSTRUCTOR: Dr. James W. Hooper
OFFICE: Corbly Hall 331A (soon, 334A)
Telephone 696-2693
email: CIS010@marshall.wvnet.edu

OFFICE HOURS: MW 2:00 - 4:00 P.M., and by appointment

REQUIRED TEXTBOOK: Sommerville, Ian, SOFTWARE ENGINEERING, 4th ed., Addison-Wesley, 1992.

CATALOG DESCRIPTION: Current techniques in software design and development using Ada, Modula-2 or C for software projects. Formal models of structured programming, top-down design, data structure design, object-oriented design, program verification methods.

PREREQUISITES: CIS 239 and CIS 320

COURSE OBJECTIVES:
* Provide a good grasp of state-of-the-art approaches to the development and maintenance of large software systems
* Provide practical experience in the team approach to software engineering
* Emphasize software requirements determination/specification, and software design.

MAKEUP OF GRADES:

Mid-Term Exam	30%
Team Project	30%
Final Exam (comprehensive)	40%

GRADE RANGES:

A: 90 and above
B: 80 - 89+
C: 70 - 79+
D: 60 - 69+
F: Below 60

LECTURE TOPICS (with references to textbook chapters)

- Introduction (Ch.1)
- Human Factors (Ch.2)
- Ada and Ada PDL--brief overview (Appendix A)
- Software Requirements
  - Requirements Definition (Ch.3)
  - System Modelling (Ch. 4)
  - Requirements Specification (Ch. 5)
- Software Design
  - Overview (Ch. 10)
  - Object-Oriented Design (Ch. 11)
  - Function-Oriented Design (Ch. 12)
  - User Interface Design (Ch. 14)
- Effective Programming (Ch. 15)

LECTURE TOPICS (continued)

- Software Reuse (Ch. 16)
- Tools and Environments
  - CASE Tools (Ch. 17)
  - Development environments (Ch. 18)
- Verification and Validation (Ch. 19)
- Software Management
  - Overview (Ch. 25)
  - Maintenance (Ch. 28, partial)
  - Configuration Management (Ch. 29, partial)
  - Quality Assurance (Ch. 31)

TEAM PROJECT

Each member of the class will be assigned to a small team (made up of three or four students) to carry out a software engineering project. The instructor will provide a handout of an assignment for the teams. Each team will organize itself relative to the assignment, conduct life cycle software activities (requirements specification, high-level design, ...), make in-class presentations on the work performed, and fully document results.

ASSIGNMENTS FOR CSD 579

Graduate students will be required to carry out additional reading assignments (journal articles or book chapters), and to submit written reports and make in-class presentations on the material.

EXAMINATIONS:

Mid-Term Exam: Monday, March 1, 1993, regular class period
Final Exam: Monday, May 3, 1993, regular class period

ROUTE PLANNING SYSTEM (RPS)

CSD 479/579 Team Project
Spring Semester 1993
Dr. J. W. Hooper, Instructor

PROJECT SUMMARY

Your small software engineering firm has obtained a contract with the Transportation Department of the local municipality to plan and develop a "route planning system" for their use. Their intention is to make use of this system to determine the "best" route for vehicles to take in travelling between locations within the city. Police and fire departments would use the system, as would the local rescue/ambulance service, utilities crews, and likely many others.

In considering this system, it appears that it could become a viable commercial product for your company, provided that you develop it with sufficient generality that other cities and regions can be equally well accommodated, and with flexibility as to intended use. This generality and flexibility will be important even in the initial use, since new streets will be added occasionally, others will be closed, etc., and since the exact nature of all uses cannot be foreseen. It also appears that it is no different conceptually to represent roads in a very large region (e.g., the state of West Virginia) than within a city, so planning with this in mind should greatly increase the possible market for your product.

Representation of streets/roads/interstate highways and their characteristics is key to the success of the system. Such aspects as one-way traffic flow, distances between points, expected travel time on road segments, street addresses, road closings, current traffic jams (e.g., a train currently blocking a crossing), streets especially busy at "rush hour", streets to favor as major thoroughfares, "scenic routes", etc., must be representable. Flexibility must be provided to add unforeseen needed characteristics for road segments.

Ease-of-use of the system is critical to its success, so "human factors" must be seriously considered in determining the user interface. The typical user will be waiting for a response, so speed of execution will be important. The fundamental functionality of the system will be to determine a route from point A to point B. Means to represent this information should be carefully considered; e.g., showing it in a color graphics display; or providing step-by-step instructions ("take third avenue westerly to eighth street east, turn left, proceed to eighth avenue, ..."), etc. Consideration should also be given to the possibility of a route description being supplied as processable input to a user's software (e.g., to software used to model the handling of hazardous waste, for which appropriate route selection is only one aspect, with other aspects including detailed representation of material handling, disposal, etc.).

Functionality should also be provided to represent certain constraints or guidance; e.g., certain street(s) that must be avoided; maximum travel time permitted; two stops to be made--give "best" routing to points B and C, in either order. "Best" route could mean shortest, fastest, most scenic, all four-lane roads, ..., and should be specifiable by the system user. In addition to "end user" functionality, means must also be provided to easily alter the representation of streets and roads, and their characteristics, maintained in the database. Capability to alter the database should be limited to a database administrator.

In planning the system, it will be necessary to determine hardware configurations to accommodate the system, perhaps with variations for the nature of eventual use (e.g., a portable PC for mobile use, with some necessary limitations on territory size and nature of user interface; an upper-end PC or workstation for large territories, color graphics output, etc.).

The first step in carrying out the project is to identify and document a candidate set of system requirements, guided by your understanding of what the municipality needs, and what other potential customers for the resulting product may need. It should be noted that the municipality has no interest in what features other prospective users of the system may want, and will not be willing to be adversely impacted by planned generality/flexibility of the system.

After the requirements are reviewed by municipality representatives, and altered as necessary, system and software design will take place, followed by implementation, testing, and installation of the initial system.

SPECIFIC PROJECT TASKS

Each team is to perform the following tasks:

* Determine a schedule for carrying out this project, and submit to the instructor; include proposed dates for reviews and submission of documents
* Determine and Specify Requirements (carefully document; include any feasibility assessments; include description of any trade-off studies, with rationale for decisions reached)
* Conduct a Requirements Review (in-class presentation)
* Conduct System and Software Design
  * Determine allocations to hardware and software (and document; justify allocations; include any trade-off studies)
  * Design the User Interface (and document)
    (Possible activity--not required: prototype some aspects of the User Interface, to help assess candidate approach(es))
  * Do high-level software design (to the module level; document; include any trade-off studies on design alternatives)

* Conduct a Preliminary Design Review (in-class presentation)
  * Do detailed software design (Using Ada/PDL; document)
  * Conduct a Critical Design Review (in-class presentation)
* Submit a final written report; include in it all the above-required information. Document changes resulting from reviews, and the reason for each change. Also describe the "dynamics" of your team--i.e., the role of each individual team member, by name, and why the work was so allocated; describe how the team functioned collectively, the decision-making process, the way team meetings were conducted, etc. Include "lessons learned"--including successes, and "if we had it to do over again, we would ..." The Final Report is due at the last regular class period.
* Give an oral report to the class, presenting an overview of the team's "dynamics" and "lessons learned"

A LIBRARY SYSTEM TO SUPPORT SOFTWARE REUSE

CIS 479 Team Project
Spring Semester 1992
Dr. J. W. Hooper, Instructor

BACKGROUND

Software engineering tools and methodologies are aiding and expediting generation of quality software, and considerable progress has been made since the inception of software engineering in the late 1960s. However, demands for software are growing much more rapidly than our ability to create it. One unfortunate aspect of the situation is that much software is being developed that need not be, if only efficient ways of reusing existing software were available. Estimates are that as much as 85% of the software in some organizations is being re-created--to say nothing about duplication between organizations. The re-creation of software carries with it the usual implications of cost, delays, and introduction of errors.

Considerable research activity is underway in software reusability, and interest is growing in obtaining efficient methods, in view of the potential high pay-off if effective reuse methods were in place. The U.S. government and many companies are beginning to undertake experimental projects in software reuse, which will take several years to reach "payoff", due to the magnitude of development involved. The assumption underlying this project is that significant improvement can be achieved in the short range by use of a system consisting of a library of carefully-selected components, with effective means for locating library components corresponding to specified criteria, and retrieving a component for reuse. A few commercial tools are beginning to appear, to help software engineers find potentially-useful existing components, and to compose a system from the reusable components. The emphasis is on libraries of well described, useful, and reliable components, and the availability of environments to help programmers find and understand existing software components.

PROJECT GOAL

The overall goal of this project is to specify and design a system for retaining potentially-reusable software components, for searching the library of components for a component (or components) meeting specified criteria, and for retrieving a component for subsequent use.

FUNCTIONAL OVERVIEW OF THE SYSTEM

1. Component Library

At the heart of the software reuse system will be the Component Library, a repository of components deemed worthy of retention for potential reuse. Each component will include a number of mandatory descriptive items in an online database, with uniform item formats for the components. Each component is expected to contain the following items, among others:

1. Name
2. Taxonomy class / subclass
3. Keywords
4. Brief technical summary
5. Method of solution
6. Rationale for choice of solution method
7. Requirements Specification
8. High-Level Design
9. Detailed Design
10. Source code / language used
11. Input Specification / Preconditions
12. Output Specification/ Postconditions
13. Observed characteristics (size, speed versus input, accuracy versus input)
14. Developer(s) of the component
15. Individuals who have used the component, their feedback on errors, degree of satisfaction
16. Test cases / results
17. Environment required (special hardware, software, communications)
18. Restrictions

Note that this information could be organized in various ways. For example, the entries could all be contained in a single file element, or a header element could be used which contains pointers to other file elements. Also note that a requirements specification could correspond to multiple detailed designs (e.g., for individual subsystems), and that a high-level design would likely correspond to multiple detailed design components. It could be that components could be included in which different solution methods are used for the same requirement--for reasons of required accuracy at the expense of greater or lesser processing time, etc. The approach taken should also accommodate updating of the entries; e.g., when user feedback is obtained.

The nature of the reusable components themselves is very important--e.g., their size and range of functionality, assumptions about non-local referencing and the nature of component interfaces, etc. We will discuss some of the issues in class, but they are not a primary concern of this project. I.E., you may take the view that you are creating this library system for the use of others, and it is their decision as to what stipulations to place on the components in the library.

Organization of the Library is an important issue, especially as it pertains to the search process, based on specified criteria. In order to achieve efficient search for Library components, it is necessary that the capability be available to specify a taxonomy for available components -- i.e., a classification scheme with classes, subclasses, etc. For example, classes could pertain to an application area (accounts payable, accounts receivable, billing, ...), to a functional area (sorting, searching, ...), etc.

2. Component Search and Retrieval

The user should be permitted to specify various criteria and characteristics, if known, in conducting the component search process. The user could directly specify component name if known, and examine the information for the component. He/she could proceed in a "browsing" mode, in which components within a specified taxonomy class are examined; or, specify a number of keywords, and be directed to components which satisy some or all of the keywords; or specify that a high-level design is to be obtained; or, specify a solution method, and a search would occur to find a component using it; or, specify the name(s) of the developer(s), or the name(s) of previous user(s); or, components implemented in a specified language -- among various possibilities; and, combinations of criteria could be specified (e.g., components for numerical integration, implemented in Ada). A simple query language should be available to the user, to ease and expedite the search process -- i.e., as a means to express the criteria for selection. A simple query language is to be designed (or an existing one chosen) as part of this effort.

Having selected a component for potential use, the user could direct that the source code be moved to his/her work area, for use "as is", perhaps, or for modification and reuse. Examples of modification could be alteration of a procedure name, or a change to array dimensions in the header (e.g., in Pascal code). The user should have the capability to retrieve other stored information for the component if desired, such as the detailed design and test cases.

3. User Interface

Considerable emphasis should be placed on an effective interactive user interface. The user will, by means of interface mechanisms, conduct the functions of component search and selection; also, components may be added to the component library. It is important to insure integrity and security of the component library. Thus effective methods must be provided to ensure that only authorized users may make use of the system, with more than one level of authorization necessary. Clearly not everyone should be able to gain access to the library; and most users should not be able to alter components in the library, remove them, or place additional components into the library--although it is necessary that authorized personnel be able to make additions, deletions, or alterations.

It would generally be that software reuse would be considered in the framework of problem solution--software design and development. Thus the user would presumably determine to use available modules, or modified available modules, to satisfy certain functions, and generate new modules for other functions. Thus, the user would be generating detailed designs for the new modules, and integrating existing detailed designs (or, modified existing designs) for modules being reused. And, as code is being generated, some of it would be newly-created, while some would come from reusable components. There are practical, logistical considerations involved in these activities. Ideally, one would have a "software engineering environment" which provides integrated support for the entire life cycle, using database(s) accessible from all phases, with tools supporting the phases, and with an overall methodology. In such a setting, the ideal approach would be to integrate reusability into the methodology, and to integrate tools to support reuse. In the current project we cannot be so ambitious. However, there are some choices to be made relative to the relationships of the various activities of the reuse environment. For example, should such activities as component examination and alteration be done by direct invocation of a text editor, after having located a candidate component (or components), or should it be possible to invoke an editor as one option from a "reuse screen". And, similar questions relate to how to effect integration of reuse source code into a work file of source code being developed, etc. Consideration should be given to such questions, and decisions made and documented, with the rationale being given also.

4. Hardware Considerations

No pre-determined computing hardware is assumed. Comparative cost considerations for hardware should have some emphasis. When considering interactive interface design, consideration should be given to the effectiveness of mechanisms for user interaction, for example (e.g., mouse, light pen, keyboard ...). Specific decisions are not necessary as to commercial hardware components, but necessary characteristics should be determined and documented. If a certain device is deemed especially appropriate for use, it should be so stated, with rationale.

SPECIFIC PROJECT TASKS

Each team is to perform the following tasks:

* Determine a schedule for carrying out this project, and submit to the instructor; include proposed dates for reviews and submission of documents
* Determine and Specify Requirements (carefully document; include any feasibility assessments; include description of any trade-off studies, with rationale for decisions reached)
* Conduct a Requirements Review (in-class presentation)
* Conduct System and Software Design
  * Determine allocations to hardware and software (and document; justify allocations; include any trade-off studies)
  * Design the User Interface (and document)
    (Possible activity--not required: prototype some aspects of the User Interface, to help assess candidate approach(es))
  * Do high-level software design (to the module level; document; include any trade-off studies on design alternatives)
  * Conduct a Preliminary Design Review (in-class presentation)
  * Do detailed software design (Using Ada/PDL; document)
  * Conduct a Critical Design Review (in-class presentation)
* Submit a final written report; include in it all the above-required information. Document changes resulting from reviews, and the reason for each change. Also describe the "dynamics" of your team--i.e., the role of each individual team member, by name, and why the work was so allocated; describe how the team functioned collectively, the decision-making process, the way team meetings were conducted, etc. Include "lessons learned"--including successes, and "if we had it to do over again, we would ..." The Final Report is due at the last regular class period.
* Give an oral report to the class, presenting an overview of the team's "dynamics" and "lessons learned"