

2

AD-A214 663

PROCEEDINGS OF THE
FOURTH ANNUAL
Ada SOFTWARE ENGINEERING
EDUCATION AND TRAINING
SYMPOSIUM

Sponsored by:

Ada Software Engineering Education and Training Team
Ada Joint Program Office

DTIC
S ELECT
NOV 21 1989
CS

J.W. Marriott Hotel
Houston, Texas
June 13-15, 1989

Approved for Public Release:
Distribution Unlimited

89 11 20 019

The views and opinions herein are those of the authors. Unless specifically stated to the contrary, they do not represent official positions of the authors' employers, the Ada Software Engineering Education and Training Team, the Ada Joint Program Office, or the Department of Defense.

ASEET TEAM MEMBERSHIP

Ms. Wanda B. Barber USA Information Systems Software Development Center-Lee Stop L-75 Fort Lee, VA 23801	AV 687-1722 804-734-1722 wbarber@ajpo.sei.cmu.edu
Captain Roger Beauman Software Engineering Training Branch 3390 TCHTG/TMKPP Keesler AFB, Mississippi 39534-5000	601-377-3728 AV 868-3728 rbeauman@ajpo.sei.cmu.edu
Captain Eugene Bingue HQSAC/SCRT Offutt Air Force Base Nebraska 68113-5000	402-294-2545 AV 271-2545 SAC.INSTRUCTOR@E.ISI.EDU
Captain David A. Cook 1403 Francis Drive College Station, TX 77840	409-693-3881 dcook@ajpo.sei.cmu.edu
Skip Dane Assist. Professor Computer Science (M.S. 9F) U.S. Naval Academy Annapolis, MD 21402	301-267-2797 dane@USNA.MIL
Mr. Leslie W. Dupaix Hill AFB, Utah 84056 USAF Software Technology Support Center OO-ALC/MMEA-1	801-777-7377 AV 458-7377
Major Charles B. Engle, Jr. Deputy Program Manager Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	412-268-6525 cengle@ajpo.sei.cmu.edu engle@sei.cmu.edu
Captain Jay Hatch US Military Academy Department of Geography & Computer Science West Point, NY 10996	914-938-2302

Captain Joyce Jenkins
US Air Force Academy
Computer Science Department
Colorado Springs, CO 80840

719-472-3530
AV 259-3530
jjenkins@ajpo.sei.cmu.edu

Major Pat Lawlis
Dept of Computer Science
Arizona State University
Tempe, AZ 85287

lawlisp@ajpo.sei.cmu.edu
lawlis%asu@relay.cs.net

Major Ed Liebhardt
AJPO, Pentagon Room 3E114
Washington, D.C. 20301-3081

202-694-0208
AV 224-0208
liebhard@ajpo.sei.cmu.edu

Ms. Cathy McDonald
IDA
1801 N. Beauregard Street
Alexandria, VA 22311

703-824-5531
AV 289-1890
mcdonald@ajpo.sei.cmu.edu

LCdr Lindy Moran
Navy Regional Data Automation
Center Code 62
Building 159, Room 335
Washington Navy Yard
Washington, D.C. 20374-1435

202-433-5236
zkc620@NARDACNET-DC.ARPA
moranl@ajpo.sei.cmu.edu

E.K. Park
Assist. Professor
Computer Science (M.S. 9F)
U.S. Naval Academy
Annapolis, MD 21402

301-267-2679
301-267-3080
eun@USNA.MIL

Major Doug Samuels
Headquarters AFSC/PLR
Andrews AFB, Maryland 20334-5000

AV 858-6941
301-981-6941
dsamuels@ajpo.sei.cmu.edu

Major Randall B. Saylor
HQ ATC/TTOK
Randolph AFB, TX 78150

AV 487-2172
512-652-2172
6426
saylor@dca-ems

Captain Michael Simpson
Software Engineering Training Branch
3390 TCHTG/TTMKPP
Keesler AFB, Mississippi 39534-5000

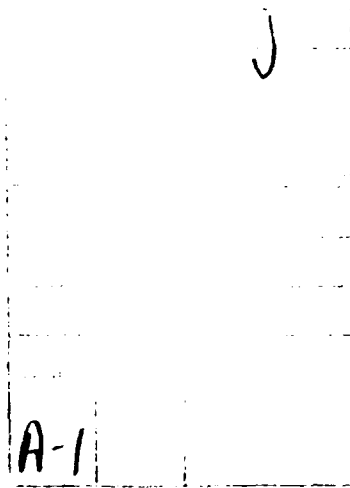
AV 868-3728
601-377-3728
msimpson@ajpo.sei.cmu.edu

Major David Umphress
Air Force Institute of Technology
AFIT/ENC
WPAFB, OH 45433

AV 785-3098
513-255-3098
dumphres@galaxy.afit.af.mil

Captain David Vega
Software Engineering Training Branch
3390 TCHTG/TTMKPP
Keesler AFB, Mississippi 39534-5000

AV 868-3728
601-377-3728
dvega@ajpo.sei.cmu.edu



v/v

This Page Left Blank Intentionally

TABLE OF CONTENTS

	Page
Message From The ASEET Chair	1
Wednesday, June 14, 1989	
Software Maintenance Exercises for a Software Engineering Project Course - <i>Charles Engle, Gary Ford, Tim Korson</i>	3
Software Engineering and Ada Database System (SEAD) - <i>Morris Liaw</i>	11
Why Ada is (Still) My Favorite Data Structures Language - <i>Michael B. Feldman</i>	17
Ada in the University: Data Structures with Ada - <i>Gertrude Levine</i>	31
Ada and Data Structures: "The Medium is the Message" - <i>Melinda Moran</i>	37
A Healthy Marriage: Ada and Data Structures - <i>Harold Youtzy, Jr.</i>	59
Teaching Old Dogs New Tricks - <i>Tina Kuhn</i>	65
Process Control Training for a Software Engineering Team - <i>Ed Yodis, Sue LeGrand, Ann Reedy</i>	73
Ada Training At Keesler Air Force Base - <i>Roger Beauman</i>	87
Integrating Ada into the University Curriculum: Academia and Industry: Joint Responsibility - <i>Kathleen Warner, Russell Plain, K. Warner</i>	99
Making the Case for Tasking Through Comparative Study of Concurrent Languages - <i>Michael B. Feldman</i>	111
Teaching the Ada Tasking Model to Experienced Programmers: Lessons Learned - <i>John Kelly, Susan Murphy</i>	121
Ada: Helping Executives Understand the Issues - <i>David A. Umphress</i> ..	135

TABLE OF CONTENTS
(continued)

Page

Thursday, June 15, 1989

The Pedagogy and Pragmatics of Teaching Ada as a Software Engineering Tool - <i>Melinda Moran, Charles Engle</i>	147
Incorporating Ada Into a Traditional Software Engineering Course - <i>Albert L. Crawford</i>	161
Motivation and Retention Issues in Teach Ada: How Will Students Learn Software Engineering When Their Only Goal is Ada on their Resume? - <i>Mary Wall, Barbara Koedel</i>	171
Use of Programs and Projects to Enhance a Software Engineering Using Ada Course - <i>Robert C. Mers</i>	177
Software Design with Ada: A Vehicle for Ada Instruction - <i>Orville E. Wheeler</i>	181
Transitioning to Engineered Ada in the Small Liberal Arts College - <i>Ronald H. Klauswitz</i>	191
A HyperCard Prototype of a CASE Tool in Support of Teaching the Berard Method of Object Oriented Development in an Undergraduate Course - <i>Frances L. VanScoy</i>	195
Authors Index	215

**MESSAGE FROM THE ADA SOFTWARE ENGINEERING
EDUCATION AND TRAINING (ASEET) TEAM CHAIR**

Major Doug Samuels, USAF

It gives me great pleasure to welcome all of you to this year's ASEET symposium. The team has worked very hard to make this fourth symposium our best. This year's competition for papers was very fierce. We received many more papers than we had time to present. They were all very good and the selection was tough. I'm sure you will find the papers and panels stimulating and thought provoking. The papers represent a broad spectrum of government, industry, and academia. Interaction and a forum for exchange of ideas is essential for the successful instantiation of Ada in these domains. I believe you'll find this symposium an excellent opportunity to accomplish these goals. We are privileged this year to have one of the language developers as our keynote speaker, Dr Robert Firth. He will bring true meaning to the phrase "instantiation of Ada in these domains." Our exhibitors bring you education and training tools that will facilitate your move to Ada. Please take the time to examine their wares. I hope you all attend the outstanding reception they provide on Wednesday night.

In order to constantly improve the symposium, we have placed critique forms through out the symposium. Please take the time to fill these out. Again, let me welcome you to the Fourth Annual ASEET Symposium. Thank you.

Software Maintenance Exercises for a Software Engineering Project Course

Charles B. Engle, Jr.
Gary Ford

*Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213*

Tim Korson

*Clemson University
Computer Science Department
Clemson, SC 29634-1906*

Abstract: Software maintenance is an important task in the software industry and thus an important part of the education of a software engineer. It has been neglected in education, partly because of the difficulty of acquiring and/or preparing a software system upon which maintenance can be performed. This paper describes a report released by the Software Engineering Institute (SEI) that provides an operational software system of approximately 10,000 source lines of code written in Ada and several exercises based on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises. This paper describes how the Documented Ada Style Checker (DASC) is presented to the instructor and how it may be used in a project based course.

Introduction

Because many if not most computer science majors go on to careers involving software development, a project-oriented course in software engineering can be very valuable in the curriculum. One of the goals of the Undergraduate Software Engineering Education Project at the SEI is to provide instructors and students with guidelines and materials for such a course.

Toward that end, in 1987 the SEI published the technical report *Teaching a Project-Intensive Introduction to Software Engineering* [Tomayko87b]. This report identified four different models for such a course and then presented detailed guidelines for one of them, the *large project team* model. This model requires 10 to 20 students organized as a software project team, with different students playing different roles (*e.g.*, principal architect, project administrator, configuration manager, quality assurance manager, test and evaluation engineer, documentation specialist, and maintenance engineer). The instructor plays the role of project manager. The student roles are defined in Appendix 1.

With such a course structure, not every student writes code; in fact, very few of the students write code. Instead, the students experience directly or indirectly all the aspects of a software development project, and that is what makes such a course a software engineering course rather than simply an advanced programming or group programming course.

A long-standing difficulty with such a course is that there is almost never enough time to develop a piece of software from scratch and then have the students do some maintenance on it. Many instructors are unaware of the importance and methods of software maintenance, and they often do not even include the subject in their course syllabus. Even instructors who do want to teach maintenance often cannot devote the time to finding or developing a system for the students to maintain. Since software maintenance is a fact of life in the software industry, it is important for students to have experienced it and learned some of the known techniques.

The intent of this paper is to describe how a recent SEI education report can make teaching software maintenance more feasible in a software engineering project course. The SEI report provides an operational software system, called the Documented Ada Style Checker, (described below); a reasonable set of documentation for the system; and specific exercises with guidelines for the instructor. Altogether, the materials could be the basis for a semester-long course. Individual exercises might be assigned as part of other courses, including a project course based primarily on new development.

The software system is written in Ada. For instructors and students new to Ada, there is a great advantage in designing a course around maintenance rather than new development. Students are able to work on a much larger system, and thus experience many more Ada constructs, than would be the case if they had to learn the language in parallel with developing code. In general, analysis is easier than synthesis in engineering.

This paper will explain what is to be found in the education report for DASC and how it might be used. For a more complete understanding of the discussion points, readers are referred to the SEI education report [Engle89].

Software Maintenance

The term *software maintenance* is generally used to mean changing a program in order to correct errors, improve performance, adapt to a changing environment, or provide new capabilities. Some consider this to be an abuse of the term *maintenance* and suggest other terminology, including *software evolution* and *post-deployment software support*. However, the term *maintenance* is widely used and understood, so we will use it here also. In simple models of software development, such as the common *waterfall model*, maintenance is considered to be an activity separate and different from development. From a software engineering standpoint, however, it is better to view maintenance as involving the same activities as those of development (such as requirements analysis and specification, design, implementation, and testing) but performed with different constraints. The most significant of those constraints is the existence of a body of code and documentation that must be incorporated into the new version of the system. Usually the cost of modifying the existing system is less than that of creating an entirely new system with the desired new functionality. This is the fundamental justification for software maintenance. However, it is the responsibility of the software project manager to recognize when this is not the case and that the existing system should be retired and a new system produced. Swanson defines three categories of software maintenance [Swanson76]:

- **Perfective:** modifications requested by the user (usually because of changing or new requirements) or by the programmer (usually because of the discovery of a better way to implement part of the system).
- **Adaptive:** modifications necessitated by changes in the environment in which the program operates (including transporting the program to a different computer system).
- **Corrective:** modifications to correct previously undiscovered errors in the program.

The exercises in the Report include some in each of these categories. There are relatively few techniques or methods specifically for software maintenance as compared to new software development. There are, however, a few software engineering techniques whose usefulness can be demonstrated especially well through maintenance efforts. Four that we recommend to instructors and students are:

- Software Configuration Management
- Regression Testing
- Code Reviews
- Stepwise Abstraction

Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. The students should be required to develop and adhere to a software configuration management plan as part of the course. The software system described in the Report consists of approximately 10,000 lines of code (in 63 separately compilable program units) and nine documents. When the students are working on the changes to both program and documentation, especially when different students are working on different changes, careful configuration management is essential to the project. Therefore one of the first recommended exercises is the development of the configuration management plan. Additional information on configuration management may be found in [Tomayko86] and [Tomayko87a].

Regression testing is defined as "selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements" [IEEE83]. Some of the exercises require major changes to the software system and therefore call for substantial retesting, perhaps involving the entire test suite. Other exercises require rather minor changes, and a single, simple retest may be sufficient. One of the first recommended exercises is the development of regression test plans. Additional information on regression testing may be found in [Collofello88b].

Code reviews offer an opportunity for software developers to discover errors or inefficiencies in their code earlier in the development process. Their use is an application of a fundamental principle of engineering: it is almost always less costly to find and correct an error early in the process rather than late. They are becoming increasingly common in industry, so students should learn at least one form of review in a software engineering course. Reviews can be conducted in a number of different ways; a good introduction for the instructor may be found in [Collofello88a].

Stepwise abstraction is a technique pioneered by IBM Federal Systems Division (now Systems Integration Division). It is used to recover the high level design of a system in the absence of design documentation. The design can then be used to plan program changes. Britcher and Craig describe the process as follows [Britcher86]:

"From the source code, the designer abstracted the module design and recorded it using PDL [Process Design Language]. Choosing the level of abstraction based on the module, the designer determined the change required. Often this was an iterative process; the designer abstracted a detailed design from the code, then generated another less detailed (yet still precise) abstraction from that design. The iteration continued until the designer was comfortable with the level of abstraction."

Some of the exercises in the Report can take advantage of this technique. For the DASC system, which is reasonably well structured and makes good use of Ada packages, this abstraction is quite straightforward. However, because it is a powerful and useful technique, we strongly recommend using it. The instructor may want to lead a classroom discussion to introduce the process.

The Documented Ada Style Checker (DASC)

Background. In 1987, Professor Jim Tomayko produced a technical report entitled *Teaching a Project-Intensive Introduction to Software Engineering* [Tomayko87b] that detailed his experiences in teaching such courses at Carnegie Mellon University and at Wichita State University. He included in his report, as appendices (but provided separately), the documents that he provided to the students and some of the reports that were produced by his students.

Using Professor Tomayko's technical report as a guideline, Professor Linda Rising of Indiana University - Purdue University at Fort Wayne (IPFW) taught a project course at IPFW designed to teach software engineering. In order to impress upon her students that this was not just an advanced programming course, she created a course that taught the principles of software engineering without writing any code. She did this by selecting an artifact, an Ada Style Checker, from the Ada Software Repository (SIMTEL20) and providing it to the students. The students were then charged with creating the requirements documents, design documents, test plans, maintenance plans, etc. When her course was finished, Professor Rising donated all of the materials produced by her students to the SEI Education Program. Since the Ada Style Checker now included documentation, its name was changed to the Documented Ada Style Checker or DASC.

When the members of the SEI Education Program received this material they tried to decide how to best use what Professor Rising had provided. They, notably Gary Ford, conceived of the idea of providing small artifacts with specific exercises for use in project courses. Dr. Ford asked a visiting scientist, Professor Tim Korson from Clemson University, to develop some exercises based upon the IPFW materials.

Dr. Korson conceived of the idea of providing maintenance exercises for the IPFW material and produced a series of such exercises that could be sensibly built from Professor Rising's students' work. He also ported the software system from the DEC VAX, on which it was provided, to a PC using the Alsys PC AT Ada compiler.

Chuck Engle, an Army Resident Affiliate to the Education Program at the SEI, then ported the system to the Ultrix operating system using the Verdix Ada compiler. Additional maintenance exercises were added by Gary Ford and Chuck Engle.

A final educational material report was issued for the DASC by the SEI in February 1989 [Engle89]. The DASC system has since been ported to the PC using the Meridian AdaVantage compiler.

Description of the DASC. The Documented Ada Style Checker (DASC) software system examines syntactically correct Ada programs and reports on their adherence to predefined style conventions. Examples of the style conventions examined are:

- Case of characters in reserved words and object identifiers
- Consistency of indentation to show program control structures
- Use of blank lines to set off program blocks
- Subprograms too short or too long
- Control structures or packages nested too deeply

- Use or lack of use of Ada-specific features

The style checker produces two kinds of reports, called a *flaw* report and a *style* report. The former identifies specific statements in the program that violate style conventions, and the latter is a quantitative summary of the program's style.

The DASC consists of an operational piece of software of about 10,000 lines of Ada code and its associated documentation. The documentation is student generated, as detailed above, and consists of the following documents:

1. Requirements Document
2. Preliminary Design
3. Detailed Design
4. Documentation Standards and Guidelines
5. Coding Standards
6. Quality Assurance Plan
7. Test Plan
8. Configuration Management Plan
9. User Manual

The requirements and design documents (items 1-3), having been produced from the source code, clearly are not as complete as one would expect in a real software development project. However, they do provide a starting point for maintenance exercises, including maintenance of the documents themselves.

The documentation and coding standards and the three plan documents (items 4-8) were used by Prof. Rising and her students to guide their project. These documents reflect both development and maintenance activities. Some of the first exercises (described below) suggested in the SEI Report involve updating these documents to reflect new maintenance activities.

The DASC is not presented as a model of good coding style, design, or documentation. In fact, if the style checker is run on itself, it reports many problems. There are some fairly obvious design improvements that could be made. The documentation is reasonable although not complete, and no formal analysis, design, or documentation technique is used.

In summary, one might say that this artifact seems to be a fairly representative example of existing software systems. This is not necessarily bad, because a valid educational objective might be to expose students to "the real world."

Use of the DASC Materials

Introduction. The SEI Report is a description of the DASC packaged with a series of suggested maintenance exercises. Discrepancy reports and change requests are provided that suggest a graduated level of difficulty in the maintenance exercises. A description of how to conduct a code review, including references to modules in the DASC that are particularly well suited for this exercise, are also suggested. Finally, the Report includes a suggested exercise for the development of regression test plans, which could lead the instructor into a discussion of the use of regression testing in maintenance.

Discrepancy Reports. The Report includes five discrepancy reports that describe known discrepancies in the DASC. These are designed to be given to the students to demonstrate *corrective* maintenance. The students should present the discrepancy reports to the configuration control board of the project team for

a determination of what is needed to correct the problems. The Report provides the instructor with a complete description of the discrepancy as well as full information as to what is causing the problem, where the error is to be found, and suggested ways of correcting the discrepancy. The instructor is then free to choose how much, if any, of this information to share with the students. An on-line copy of a blank discrepancy request is provided so that the instructor may modify the form to suit his or her specific needs.

Change Requests. There are seven change requests included in the Report. The purpose of the change request is to suggest improvements or modifications to the system that the students may perform, thereby affording the student the opportunity to conduct *perfective* maintenance. The change requests are provided in a format similar to the discrepancy reports with a copy of a filled-in change request for each of the suggested changes. The instructor is once again provided with a description of why the change is needed and some tips on how the change may be accomplished. An on-line copy of a blank change request is provided so that the instructor may make changes in the format of the form if desired.

Code Reviews. In conjunction with one of the change requests, the Report describes a portion of the DASC that is suitable for conducting a code review, i.e., it illustrates some of the principles of a code review and demonstrates its effectiveness. The instructor is referred to two other SEI documents, [Collofello88] and [Cross88], for the details of how to conduct a code review.

Suggested Course Ideas. The materials in the Report may be used in a semester long course on software maintenance or may be included in lesser amounts as part of other courses. The idea behind the Report is that the instructor can start his or her course with a medium sized piece of code that is probably beyond the capabilities of his or her students to completely understand in detail. This forces the students to work in teams and to be cognizant of the human factors in group dynamics. In addition, it rapidly demonstrates the need for configuration management and for planning. The interested instructor is referred to [Tomayko87b] for a detailed discussion of the manner in which to conduct such a course.

Summary

The SEI Report on the DASC describes a self-contained artifact, associated documentation, and a series of exercises that can impart to the student the skills needed for software maintenance in all of its various forms. It provides the instructor with a tool that has long been missing from the college curriculum; an artifact large enough to use to teach software engineering (not advanced programming), yet small enough to be used on a PC. The DASC can be used to teach students the fundamentals of software maintenance, including regression testing and code reviews. Perhaps more importantly, by providing a system too large for one student to completely master in a normal academic session (semester or quarter), the DASC can be used to establish the cognitive dissonance necessary to make students want to develop these skills.

References

- [Britcher86] Britcher, Robert N., and James J. Craig. "Using Modern Design Practices to Upgrade Aging Software Systems." *IEEE Software* 3, 3 (May 1986), pp. 16-24.
- [Collofello88a] Collofello, James S. *The Software Technical Review Process*. Curriculum Module SEI-CM-3-1.5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

- [Collofello88b] Collofello, James S. *Introduction to Software Verification and Validation*. Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.
- [Cross88] Cross, John A., editor. *Support Materials for The Software Technical Review Process*. Support Materials SEI-SM-3-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.
- [Engle89] Engle, Charles B., Jr., Gary Ford, and Tim Korson. *Software Maintenance Exercises for a Software Engineering Project Course*. Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1989.
- [IEEE83] IEEE. *Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 829-1983, Institute of Electrical and Electronics Engineers, 1983.
- [Swanson76] Swanson, E. B. "The Dimensions of Maintenance." *Proc. 2nd International Conference on Software Engineering*, IEEE Computer Society, 1976, pp. 492-497.
- [Tomayko86] Tomayko, James E. *Support Materials for Software Configuration Management*. Support Materials SEI-SM-4-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1986.
- [Tomayko87a] Tomayko, James E. *Software Configuration Management*. Curriculum Module SEI-CM-4-1.3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.
- [Tomayko87b] Tomayko, James E. *Teaching a Project-Intensive Introduction to Software Engineering*. Technical Report CMU/SEI-87-TR-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.

Appendix 1. Project Team Roles

Principal Architect: Responsible for the creation of the software product. Primary responsibilities include authoring the requirements document and specification document, advising on overall design, and supervising implementation and testing.

Project Administrator: Responsible for resource allocation and tracking. Primary responsibilities are cost analysis and control, computer and human resource acquisition and supervision. Collects data and issues weekly cost/manpower consumption reports and the final report.

Configuration Manager: Responsible for change control. Primary responsibilities include writing the configuration management plan, tracking change requests and discrepancy reports, calling and conducting change control board meetings, archiving, and preparing product releases.

Quality Assurance Manager: Responsible for the overall quality of the released product. Primary responsibilities include preparing the quality assurance plan, calling and conducting reviews and code inspections, evaluating documents and tests.

Test and Evaluation Engineer: Responsible for testing and evaluating individual modules and subsystems and for preparing the appropriate test plans.

Designer: Primary responsibility is developing aspects of the design as specified by the architect. During the pre-design stage, this person could assist in a literature search to explore similar products or problems.

Implementor: Primary responsibility is to implement the individual modules of the design and serve as the technical specialist for a particular language and operating system. During the requirements specification and design stages, the implementors could develop tools and experiment with new language constructs expected to be needed in the product.

Documentation Specialist: Responsible for the appearance and clarity of all documentation and for the creation of user manuals.

Verification and Validation Engineer: Responsible for creating and executing test plans to verify and validate the software as it develops, including tracing requirements through specification, design, coding, and testing. Also responsible for code inspections. Acts as a member of an independent group.

Maintenance Engineer: Primary responsibility is creating a guide to the maintenance of the delivered product.

The Software Engineering and Ada Database (SEAD)

**Dr. Morris M. Liaw
University of Houston at Clear Lake**

INTRODUCTION

The University of Houston/Clear Lake (UH/CL) and NASA/Johnson Space Center have cooperated on a project to increase the availability of information about Software Engineering and Ada. The result of this project is the Software Engineering and Ada Database (SEAD), an on-line relational database. SEAD was funded by NASA/JSC and developed at UH/CL. SEAD is currently available to NASA personnel and Contractors via the NASA/JSC Center Information Network (CIN).

REQUIREMENTS

SEAD was conceived as an information resource for NASA and NASA Contractors, containing information about Ada-based resources and activities which are available or underway either in NASA or elsewhere in the worldwide Ada community. The sharing of this information will reduce duplication of effort while improving quality in the development of future software systems.

To be successful, the information on SEAD must be readily available to potential users, thus SEAD required on-line access.

Since SEAD will serve users from a variety of backgrounds, located in all parts of the United States, a primary requirement was that the database be easy to use, with ample on-line help documentation.

To decrease maintenance costs, NASA required that all development be done using commercially available, well supported, and widely used tools for the database management system, menu manager, etc.

SEAD would contain information of interest to and available from the Software Engineering Ada community. It would need to be continuously updated.

DEVELOPMENT

To meet these requirements, the database structure was developed using the ORACLE Relational DBMS, running in a VM/CMS environment.

connectivity to the NASA/JSC Center Information Network (CIN).

The user interface, which is completely menu driven, was built with the ISPF Dialogue Management Server (DMS) and ORACLE's SQL*Forms.

A help-line will soon become available for users. Additional documentation is also available, such as the SEAD Administration Guide, User's Manual, SEAD Brochure, Quick Reference Chart, etc.

History

Development began in June, 1986 with the study of what information was available related to software engineering and Ada. Data items were identified, analyzed and grouped. A logical data base was constructed, then the physical data base was created. An on-line interactive retrieval system was developed to query the database. In October, 1986, a prototype demonstration was given at UH/CL. Based on the results of the demonstration, several changes were made during the next three months.

In January 1987, SEAD was installed at NASA and Alpha testing was conducted through the end of February 1987. Actual data was gathered and loaded as the system was refined. In April 1987, the first release of SEAD was finalized and released.

However, additional data is still being collected, and feedback from users is being analyzed for future improvements. The changes to be included in that release were approved in August 1987. The second release was available in January, 1988.

FEATURES

Making SEAD easy to use involved two main features: menus and on-line help. Once a user is connected to the network, he is automatically taken to the SEAD Main Menu, from which he can select which general topic he is interested in. At this point he can also get general information about the database and the use of special function keys. After the Main Menu, another level of menus, arranged hierarchically,

bring the user to the SQL*Forms panels where the data can be retrieved and displayed.

Each menu is visible on a single screen, so users can see all the selections at the same time. Also, each menu has a help selection with information about the menu and how to use it.

At the bottom level, SQL*Forms requires the use of special function keys (or sequences of keys) to enter queries and browse the data. There are too many data elements for each item to be displayed on one screen, so the elements were grouped logically, and the user can browse them one screen at a time. Help at this level includes: how to construct and enter simple queries, how to use the function keys to move from one screen to another, and a description and complete attributes of any element on any screen.

Since on-line updates can be made in any part of the database anytime the system is running, users may have access to the updates immediately.

In order to maintain the integrity of the database, key constraints, domain constraints, and data consistency are supported. The system enforces key constraints by not allowing null or duplicated primary key values. The system can validate the range and format of a field value at the time it is entered to enforce domain constraints. Deletion or insertion a record that will violate the consistency of the database is prevented by the system, and key changes are propagated automatically through triggers.

Security at the system level is handled by the use of passwords which prevent visibility of and access to other applications on the system. Users have read-only access to the database, so all updates to the data are controlled by the Data Administrator. Changes to the structure of the database must be approved by NASA management through formal change control procedures.

CONTENT

The information in the SEAD is organized into five main areas:

- 1) Education and Training
- 2) Publications and Conferences
- 3) Projects
- 4) Compilers and Other Products
- 5) Reusable Software Components.

Additional services include a bulletin board for announcements and information about meetings, and cross

referencing of information about projects publications, compilers, projects and reusable packages.

SEAD is not intended to contain full information about these items. Rather it serves as a resource, showing what is available or in progress and where to obtain more information. For example, we do not include "Lessons Learned" information on projects, but we do include the name and address of a contact person or organization wherever possible.

ENHANCEMENTS

In the next release, scheduled for March 1989, changes will be made to the organization of the menu, add a few data elements to some screens, and include some additional services. The organizational changes will mainly consist of breaking compilers out as a separate category from other projects, and regrouping the other categories into the following areas: Additional data items will include such things as a field for contact persons' electronic mail address and the validation data and version for compilers.

UH/CL specific information from SERC, the High Technology Laboratory Library, SEPEC, RICIS will also be available from SEAD.

AdaNet is a national network supported by NASA Office of Technology Utilization, the Department of Defence Ada Joint Program Office, the U.S. Dept. of Commerce, etc.

The AdaNet contract is managed for NASA by UH/CL. The experience gained by the SEAD project prepared the University personnel for the task of managing a national resource such as AdaNet.

An AdaNet prototype was just released. SEAD may become available as a menu item on the AdaNet prototype.

RESEARCH PROBLEMS

Most of the problems encountered in developing the SEAD were associated with trying to adapt commercial applications to an on-line system for a broad base of users. For example, the lack of text processing capabilities in ORACLE limited the size of data items to the width of the screen. Many technical papers have titles longer than that. In addition, SQL*Form's reliance on special function keys made the user interface difficult for users with terminals that do not have the special keys or whose function keys do not produce the expected sequence of characters.

Another issue arose from having the database located on the NASA/JSC network. The SEAD must be accessible to everyone who has a need for information about Software Engineering and Ada. In contrast, to maintain the security of their network, NASA has limiting access by requiring that passwords must be requested in writing and cannot be provided on-line.

It seems that the best solution to this problem will be to move the SEAD to a system at the UH/CL campus.

We also see the desirability of investigating other data models, such as Object-Oriented data model, for large database systems.

CONCLUSIONS

The Software Engineering and Ada Database was modeled, built to requirements, and delivered on schedule. The goal of providing access to information on Software Engineering and Ada has been met.

WHY Ada IS (STILL) MY FAVORITE DATA STRUCTURES LANGUAGE

Michael B. Feldman
Professor

Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, DC 20052

(202) 994-5253
MFELDMAN@GWUSUN.GWU.EDU

INTRODUCTION

The purpose of this paper is to present the case for Ada as a language for teaching data abstraction and data structures in an undergraduate computer science curriculum.

The choice of a programming language for this or that curricular subject must be made on the basis of a number of factors, many of which are different from those important in an industrial setting. In the case of Ada in particular, industry (or at least that segment that develops software for defense needs) operates under a mandate to use the language. Government policy decrees that Ada *will* be used. Absent a reversal of policy, industry has little choice: he who pays the piper has the right to call the tune. The "competition" is then one between vendors of Ada compilers and environments.

In the academic world, the situation is altogether different. The American college environment is subject to only minimal control and standardization, even in technical fields. Faculties are notoriously contentious; decisions are often made by committee, not by management; budgets are always stretched thin and costs cannot easily be passed on to the "consumer." Since no government mandate exists for its use in the university setting, Ada must be justified on the merits, in the face of a faculty's own unique brand of inertia. Ada proponents—including compiler vendors—must realize that the competition is between Ada and other languages, not between Ada vendors X and Y.

THE GWU EXPERIENCE

CSci 159, *Programming and Data Structures* is an undergraduate course in the George Washington University Department of Electrical Engineering and Computer

Science, required for undergraduate majors in computer science and computer engineering. The course is also populated by would-be graduate computer science majors who have a weak background in modern data structures, and by graduate students from other fields. Typical enrollment is in the neighborhood of 100-150 students per year.

Ada has been the primary language taught in CSci 159 since academic year 1983-84; since AY 1985-86 all students have coded in Ada as well. Compilers used include Verdix VADS under Unix and VMS, TeleSoft TeleGen2 under VMS and VM/CMS, and Meridian AdaVantage under PC-DOS. The primary text is this writer's *Data Structures with Ada* [Reston/Prentice Hall 1985]; various Ada language textbooks have been used as collateral reading.

Six years of experience, with perhaps 600 students involved, confirms our view that in the family of widely-available procedural languages, Ada embodies by far the most effective collection of features to facilitate the teaching of data structures. We shall present this view with reference to other candidate languages, specifically standard (ANSI) Pascal, Turbo Pascal¹, Modula-2, and C.

STRENGTHS OF Ada FOR TEACHING DATA STRUCTURES

The course in question fits into our curriculum at about the sophomore level; the students have typically had a semester or two of Pascal. The emphasis in the first two courses is necessarily on program control and algorithm development, and the whole complex of issues we call "structured programming." The primary focus in a third course should be on data abstraction.

Ada supports data abstraction far better than "the competition" in a number of ways. Chief among them are

- functions can return structured objects, not just scalars;
- packages impose a separation of specification and body;
- private types exist and there is no restriction on the type classes which can be made private;
- arrays can be "conformant" (to use Pascal terminology) in all dimensions.

¹ A serious question of principle is whether, in this age of portability concerns, a single compiler vendor should be able to define the *de facto* standard for a programming language. This is, of course, a matter of taste; our own position is that it should not. Using, for example, Turbo Pascal sends a message to our students that portability and standardization play second fiddle to bells and whistles. We discuss Turbo Pascal in this paper because it is, unfortunately, so popular.

Function result types: Ada. That a function may return a value in any type class, including specifically a record or array, is a feature about which little fuss is made in the Ada literature. But it makes a big difference. Consider the standard example of a rational type:

```
type Rational is record
    Numerator: Integer;
    Denominator: Positive;
end record;
```

Each object of this type is a record. In languages with unrestricted function return values, one can define operations of the form

```
function Add(left, right: Rational) return Rational;
function Mult(left, right: Rational) return Rational;
```

and given four objects R1, R2, R3, R4, of type Rational, one can write statements of the form

```
R1 := Add(R2,R3);
```

or even use functional composition, say,

```
R4 := Mult(R1,Add(R2,R3));
```

The advantage of this functional notation and composition should not be underestimated: many applications require manipulation of programmer-defined mathematical structures and the notation used by programmers should model as closely as possible the notation used by mathematicians and engineers. If Ada did not allow functions to return structured types, our operations would have to be procedures, e.g.

```
procedure Add(Result: out Rational; left, right: Rational);
procedure Mult(Result: out Rational; left, right: Rational);
```

and a use of the operation would be written as a procedure call, which cannot be composed. Our nice composed expression above would have to be written

```
Add(TemporaryResult, R2, R3);
Mult(R1, TemporaryResult);
```

which surely does not look mathematical.

A work-around in Pascal and Modula-2 is to pass *pointers* to the structured objects as function arguments and results. This technique creates problems such as aliasing and dynamic allocation. Such excessive use of pointers is poor software engineering; it is also difficult to explain to students why it should be necessary.

We note that Ada also provides for operator symbol overloading, so that e.g.

```
function "+"(left, right: Rational) return Rational;  
function "*" (left, right: Rational) return Rational;
```

is permitted, with corresponding use

```
R4 := R1 * (R2 + R3);
```

making for a *very* mathematical-looking expression. This feature falls into the category of convenient "syntactic sugar;" we think it is less fundamental or necessary than the unrestricted function return value.

Ada also allows array objects to be returned from functions, so that one can write and use vector and matrix operations very conveniently and intuitively. This is related to the general Ada array capabilities, about which more below.

Function result types: the Competition. *Standard Pascal* does not permit records or arrays to be returned from functions. Neither do the Pascal derivatives *Turbo Pascal* and *Modula-2*. The proposed C standard allows records—but not arrays—to be returned. In the present example, C would allow the rational type but not the vector or matrix.

Ada's unrestricted function return values makes Ada compilers undoubtedly more difficult to implement; we think the price is worth paying.

Packages: Ada. The separate package specification introduces the student to the idea of a "contract with the user." Students trained in (standard) Pascal tend to focus on "getting an answer" rather than "building a product." Using packages encourages a student to design a software component and carefully implement this contractual relationship with the component's user. The contract idea is reinforced by the separation of specification and body into separate files, separately compiled: students can see clearly that if something is not written in the spec, it's not visible to a client. Separate compilation means that programs dependent on a package need not be re-compiled if only the body, not the spec, is changed.

In CSci 159, programming assignments often require just the building of a package, with *no* client program at all except a test driver to validate the package. This is often not easy for students whose intuition drives them to focus on pretty interfaces and *getting an answer*, as opposed to developing a component intended for use by *another programmer* and not an end user. The grading system for projects must place heavy weight on the contractual relationship: the contract must describe *how a package is to be used*, not the details of *what it does*. CSci 159 allocates 30% of the grade to the quality of the package specification and its supporting user document.

Packages: the Competition. *Standard (ISO or ANSI) Pascal* has, of course, no notion of a package. *Turbo Pascal* provides a package-like structure called the "unit" (borrowed from UCSD Pascal), but the interface (specification) and implementation (body) must be in a single file. This diminishes the abstraction value—the student does not see the two sections as physically distinct—and also requires recompilation of dependent program segments every time something is changed, even if the change is only a detail in the implementation. A disadvantage of Turbo Pascal in general is that it is not available on Unix and other shared machines, and also that, at least until now, version k+1 has differed *significantly* from version k. And the IBM-PC and Macintosh versions are not even compatible: even if one ignores special operations for graphics, etc., there are *syntactic* differences between the two.

Modula-2 provides the *library module*, with definition (specification) and implementation (body) modules (files), separately compiled. This capability is quite similar to Ada, in spite of differences in the way import and export directives are written. Compilers are widely and inexpensively available and support a (generally) common language. A serious liability is the treatment of private types (see below).

C provides only a very rough equivalent to packages, namely the separation of groups of subprograms and type declarations into different files. Compilers are legion; the language supported is reasonably standard. Enforcement of interfaces, however, is strongly compiler-dependent.²

Private types: Ada. The private type, with its hidden implementation, is of course intimately related to the package. Ada allows any type to be made private or limited private; in particular, structured types can be private, and this forms the basis for an abstract data type scheme.

The software-component philosophy embodied in the package and the private type pays off handsomely in more advanced courses, *even if the student goes on to develop programs in other languages*. Private types are an important subject in CSci 159; we see anecdotal evidence that CSci 159 graduates who choose to use C, for example, in senior projects, write better C because of their Ada exposure.

Private types: the Competition. *Standard Pascal* provides no private types. *Turbo Pascal* allows a unit to export a type, but its internal structure is visible to clients. One could hide, e.g., the fraction record type definition in a unit whose existence is not advertised, then make the fraction type itself a pointer to the hidden record type. This dodge is unsatisfying: it requires an extra unit, spreading the code for a single abstract type into two units, and carries along all the disadvantages of pointers.

² C++, the recently-developed extension to C, provides an object-oriented programming language more similar to Smalltalk than to Ada. C++ may become an important competitor, but is not yet widely available. A disadvantage for students is the less-than-obvious syntax.

Modula-2 improves the situation, but only a bit. A private type may be declared in a definition module, but its type is *required* to be a pointer to another type declared in the implementation module. At least the code for a single abstraction appears in a single library module, but the pointer difficulties persist.

C provides no notion of a private type. A work-around similar to the one described for Turbo Pascal could be invented, but it would surely be cumbersome.

An important consequence of the generality of function results and private types is that *access types (pointers) are unnecessary except to implement linked structures*. We believe that it is inappropriate to have to trade the niceness of functional notation for the forced clumsiness of pointers, solely because of a language limitation.

Array handling: Ada. Ada provides the "unconstrained array type" for an arbitrary number of dimensions. While the *number* of dimensions of an array must be specified in the type declaration, the bounds may be left unspecified until variables are declared. Further, unconstrained array types may be used in subprograms as formal parameters and function results. This facilitates a very natural implementation of vector and matrix packages, an important application often studied in data structures courses. For example, consider a package exporting a matrix type

```
package Matrices is
  type Matrix is
    array(Integer range <>,           -- bounds left open
           Integer range <>)         -- till variable
    of Float;                         -- is declared
  ...
  function "+" (left, right: Matrix) return Matrix;
  Conformability_Error: exception;
end Matrices;
```

Here we have combined many of the capabilities of Ada: the package, the unconstrained array type, overloaded operator symbols, unrestricted function result types, and the definition of application-dependent exceptions. In the package body, below, the code for the addition operator is given. Note the use of the attribute functions *First*, *Last*, and *Range*, which give the low bound, high bound, and bounds range, respectively, for the two dimensions. The subprogram can simply ask its actual parameters what their bounds are, then operate accordingly—in the event, create a temporary matrix sized according to the bounds of the inputs, fill it with values, then return this new matrix to its caller. Given three matrix objects

```
M1, M2, M3: Matrix(-5..5);
```

then the statement

```
M1 := M2 + M3;
```

can be written in the natural mathematical style. Note in the body of the addition operator that `Conformability_Error` is raised if the addition of the two matrices would be mathematically meaningless.

```
package body Matrices is  
  ...  
  
  function "+"(left,right: Matrix) return Matrix is  
    Temp: Matrix(left'range(1), left'range(2));  
    -- size of result gotten from size of input  
  
    begin  
      if left'First(1) /= right'First(1) or  
        left'Last(1)  /= right'Last(1)  or  
        left'First(2) /= right'First(2) or  
        left'Last(2)  /= right'Last(2)  
      then  
        raise Conformability_Error;  
      end if;  
  
      for row in left'range(1) loop  
        for col in left'range(2) loop  
          temp(row,col) := left(row,col) + right(row,col);  
        end loop;  
      end loop;  
  
      return temp;                                -- array!  
  
    end "+";  
end Matrices;
```

Array handling: the Competition. Neither *Standard Pascal* nor *Turbo Pascal* nor *C* has any equivalent at all to the unconstrained array type (which actually resembles a feature in PL/1). *Modula-2* provides the "open array parameter" for subprograms, in which a *one-dimensional* array parameter may be passed without knowing its bounds; there is a rough equivalent to the attribute functions in this case. But this is permitted only for one-dimensional arrays, so the ability to create a general matrix package in a natural way is severely limited.

COMMENTS

Following the body of this paper is a chart comparing, in summary form, the various features we have discussed here. Also attached is a sample syllabus for CSci 159, some sample projects, and grading guidelines.

We have concentrated here on a selected few Ada features we believe are especially useful in teaching data abstraction. We have not paid particular attention to linked data structures, as these are essentially the same in all modern languages. For brevity we have not included a discussion of generics; this subject warrants a paper in its own right.

Our undergraduate curriculum encourages students to learn a number of programming languages, because we believe that multilingual graduates are more openminded and accepting of change than those steeped in a single language with only the most superficial exposure to others.

Recently we have made the syntactic transition to Ada a bit easier by distributing a diskette of about fifty "small" Ada programs which cover the inner syntax of the language and the intricacies of the input/output libraries. Some of these programs are "booby-trapped" with deliberate compilation errors. The students are asked to compile and try these programs; if they can understand them all, including the reasons for the various errors, they know the rudiments of the Ada "Pascal subset" and are ready to dive into writing packages. These small programs also serve as templates for writing other programs, especially those using various kinds of input loops.

Our students grumble a bit, at first, about being required to learn a new language for CSci 159, but they take readily to Ada once they begin to sense its power for building systems. Once students have picked up the rudiments, they often comment that syntactically, Ada is *easier* than Pascal; we tend to agree. And increasingly they choose Ada for upper-division projects where they are given a choice of language. Ada is at GWU to stay.

COMPARISON OF SELECTED FEATURES
AMONG FIVE PROGRAMMING LANGUAGES

	<i>Ada</i>	<i>ANSI Pascal</i>	<i>Turbo Pascal</i>	<i>Modula-2</i>	<i>C</i>
<i>Allowed types of function result</i>	scalars; pointers; records; arrays	scalars; pointers	scalars; pointers	scalars; pointers	scalars; pointers; records
<i>Separate compilation of encapsulation</i>	package	none	unit	library module	file
<i>Compiler-enforced interface to abstraction</i>	yes	no	yes	yes	compiler-dependent
<i>Private types</i>	yes	no	no	pointers only	no
<i>Unconstrained array types</i>	yes	no	no	no	no
<i>Conformant arrays in subprograms</i>	multi-dimensional	no	no	one dimension only	no

THE GEORGE WASHINGTON UNIVERSITY
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

CSci 159 - Programming and Data Structures
Prof. M. B. Feldman
COURSE OUTLINE

WEEK	LECTURE SUBJECT	READINGS
1	Introduction; Abstraction	Chap. 1
2	Abstract Data Types	
3	Performance Prediction; "big O"	Chap. 2
4	Arrays and their representation	Chap. 3
5	Sparse Arrays	
6	Linked Lists	Chap. 4
7	MIDTERM EXAMINATION	
	Stacks and Queues	Chap. 5
8	Graphs	Chap. 6
9	Midterm exam post-mortem; Trees	Chap. 7
10	Trees continued	
11	Hash Table Methods	Chap. 8
12	Sorting	Chap. 9
13	Sorting (continued)	Chap. 10
14	Review	
15	FINAL EXAMINATION	

THE GEORGE WASHINGTON UNIVERSITY
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

CSci 159 - Programming and Data Structures
Prof. M. B. Feldman
GENERAL ORGANIZATION OF COURSE

CSci 159 is organized around *readings, lectures, programs, and examinations.*

The *readings* in this course are, it is hoped, interesting and useful. The readings use Ada. The basic syntax of Ada is very similar to Pascal; the more interesting Ada concepts will be taught in this class. *If you don't know any Pascal, you can probably skip the intermediate step and go directly to Ada, but you'll have to do a lot of reading on the side.*

Attendance at *lectures* is, in principle, required, but you do not need special permission to be absent. However, you are responsible for whatever happens in class, so it's in your interest to come regularly. *Little sympathy will be shown to a student who finds him(her)self in trouble after having routinely cut class!*

There will be three (perhaps four) *programming assignments*. Details will be given to you separately, but, in summary, they will be written in *Ada*. In principle you will use the GWU computing facilities. You may also use a computer at home or at work. If you have an IBM PC or compatible and wish to purchase an Ada compiler (about \$100. direct from the vendor), I can supply you with information.

Programs must represent the results of your own work. I cannot prevent your speaking with friends to sketch out a solution. But if you collaborate on the detailed design or coding, or copy a program from an acquaintance, then submit the results as your own work, *I will charge you with plagiarism, and I will win.*

Programs submitted by 6 PM, one week following the due date, will be deemed to be on time. This is a "grace period" to allow for unexpected computer problems, illness, etc. You do *not* need special permission to turn a program in late. However, *programs turned in after the grace period will be subject to a grade penalty of 20% per week.* This policy is to encourage you to spread your time commitment and your demands on the computer evenly over the semester, and will be enforced.

There will be two *examinations*. The *midterm exam* will be 1-1/4 hours long; the *final exam* will be 2-1/2 or 3 hours long. Examinations are *open book*.

Your *grade* in the course is based on these weighting factors: *midterm exam 25%; final exam 45%; programming projects 30%.* Conversion from numerical to letter grades is done *only* when final grades are to be assigned. Grades are usually adjusted to the overall performance of the class.

THE GEORGE WASHINGTON UNIVERSITY
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

CSci 159 - Programming and Data Structures
Prof. M. B. Feldman

GUIDELINES FOR REPORT PREPARATION

Each programming assignment is graded on the basis of 0-10 points: *four points* for correctness of program and test results; *three points* for internal coding style, comments, etc.; *three points* for user documentation.

The grader will ask the following questions, *in this order*: (1) What capabilities are promised in the user guide? (2) Do the test results show that the program delivers on all its promises? (3) Is the source code well-organized and appropriately and courteously documented?

Test Results: Your tests are required to *demonstrate convincingly that your programs perform as advertised*. Your test program does *not* need to produce elaborate or beautiful output; annotation by hand is acceptable. But *your test results must be self-contained; you cannot get full credit if the reader has to look at your source programs to interpret them*.

Program listings and Internal Code Style : Follow the style guidelines you've learned in the past for program layout; these are probably acceptable. Use plenty of white space, particularly to set off one subprogram from the next. Try not to have a subprogram broken in the middle over two printed pages (some compilers make this difficult, but do the best you can). It is much easier to read a program when it's all on one page. In the Feldman book, great care was taken in the layout of the programs; the book can serve as a good example for you. Functional comments are important, but should not be over-used.

User Guide and other external documentation:: Abstraction and information-hiding are important principles in this course. The *user* of your program needs to know what it does, not how it works! If the project is a package, a brief but complete description of the interface to each routine is what the user needs. Do not write too much, or reveal detail that the user does not need to know, or you will lose points!

General Format: Use an editor or word-processor. *You will lose points for a report which is hard to read!* Get laser-printer output if you can. Submit everything in a letter-size (8.5" x 11") report cover, preferably a simple, lightweight, and inexpensive one.

Last words: An important part of any real-world programming project is the quality of the packaging and the thoroughness of the testing. If you're careful, follow these guidelines, and *don't leave your documentation until the very end*, you will have no problems with the grades you get.

CSci 159 - Programming and Data Structures Sample Programming Projects

Project 1— Sets.

Chapter 1 of the Feldman book describes a package to handle the abstract data type `NaturalSets`. Your programming assignment is to implement, in Ada, a package to handle sets drawn from the universe 'A' .. 'Z', using the bit-map approach which represents a set as an array of booleans.

You will need to write and test all the necessary functions and procedures for dealing with sets. You will also need to design a test program whose *only* purpose is to show that the various routines in your package “deliver on what they promise.” Please take care to design a thorough and effective set of tests for your package. In this first assignment, many students end up with lowered grades because their tests are inadequately thought out. Often test cases are chosen arbitrarily or too many identical tests are chosen. Be prepared to *justify* why you chose each test case! Also, your test output must be “self contained” - the grader will *not* read your source code in order to understand your tests. Your test run may be annotated by hand, but it must be annotated. If the grader cannot understand your test without reading your source code, your grade will be lowered.

Your package also needs a *user guide*, which consists of a brief description of each user-callable function or procedure, and some discussion of how the user is to create sets. Do *not* reveal to the user any information intended only for the maintainer!

Project 2 — Graphs and Linked Lists.

In this project you will develop a package for *directed graphs*. Figure 6-8 of the Feldman book suggests an implementation using linked lists; your task is to design, code, and test a package to do graph operations using this scheme. Among the operations should be:

- initialize a directed graph (let the node names be positive integers)
- insert an edge into a graph
- delete an edge from a graph
- determine whether there is an edge from node *x* to node *y*
- traverse a graph using `DepthFirstSearch` (Figure 6-13, where `Visit` simply prints out the name of the node visited)
- traverse a graph using `BreadthFirstSearch` (Figure 6-15)

Note that `BreadthFirstSearch` requires the use of a queue; you should build a package to work with queues and then import it. Figure 5-9 gives you a start on this.

Note also that both traversals require the use of sets. Funny thing: you built a

package for sets in project 1! Just modify it to work with the right kind of set elements (essentially go back to `NaturalSets`).

Project 3—Benchmarking sort algorithms.

This project involves benchmarking three sort algorithms to compare their theoretical big O running time with their actual times as measured on the Sun, the Vax, or the IBM PC. Procedures for Bubble Sort ($O(N^2)$) and Heap Sort ($O(N \log N)$) can be found in Chapter 9. We will give you files containing these sorts; your task is to prepare a third sort procedure implementing Shaker Sort (2-way Bubble Sort), then compare the actual running times of all three for different array sizes.

Each of the three procedures is to be run on arrays of size 2^K where K runs from 4 (i.e. array size 16) to 10 (i.e. array size 1024). Three arrays of integers will be used: one in upward-sorted order, one in downward-sorted order, and one in random order. You will be given a random 1024-element array; note that you can copy "slices" of this array to use in your sort procedures. For the sorted arrays, use the subscripts themselves as values, i.e. if the array has N elements, use $A(i)=i$ for the upward sorted case and $A(i)=N-(i-1)$ for the downward sorted case.

How you organize your driver program to do all the sorting runs is up to you; note that there will be a total of 63 sorts performed: 7 sizes \times 3 array orderings \times 3 sort procedures. To save on machine overhead, try to combine a number of sort runs into a single run of your program (perhaps all 21 runs of a given sort procedure, for example). The coding for the project is very easy. A large part of the project is the way in which the results are presented. Present your results in tabular form (the table does not have to be produced by your program) and as a set of appropriately designed graphs.

The Ada standard provides for "time of day" operations but not for CPU-time operations, so on a multi-user machine one needs a separate package to measure CPU times. We will provide code and documentation for this package. On the PC, since it is a single-user machine, the built-in Ada timing operations could be used, but we will supply a package for the purpose, for compatibility with the other systems.

ADA IN THE UNIVERSITY
DATA STRUCTURES WITH ADA

Gertrude Levine
Assistant Professor
Fairleigh Dickinson University
Teaneck, New Jersey

1. INTRODUCTION

The choice and implementation of data structures are critical for the clarity, correctness, and effectiveness of algorithm design. A course in Data Structures is therefore fundamental to most Computer Science programs. The programming language Ada is particularly effective for the presentation of such material.

Many Computer Science curriculums have adopted Pascal as their introductory programming language. The set of facilities provided by Pascal is relatively small, and the language's unity of design and structure are helpful to beginning programmers. Pascal's control structures are adequate for algorithmic expression. In addition, Pascal compilers and support systems are relatively inexpensive, and many students are able to explore and experiment with concepts at home.

Once Pascal is mastered, many of its features, such as scope rules and pointers, are so similar to Ada's that the transition from a freshman's Pascal to a sophomore's Ada is relatively painless. Thus the material contained in a course in Data Structures can be presented using the Ada language, which is more appropriate for the implementation of abstract data types.

2. ADA FOR DATA STRUCTURES

Ada's package construct is a powerful facility for the encapsulation of a data type. A package may contain a type definition, constants or variables of the type, operations on the type (represented by functions and procedures), and conditions under which the operations and/or the implementation fail (represented by exceptions) in a specification unit that is visible to other program units. The specification provides an interface to users, so that they can manipulate objects of the data type by the operations supplied. If the data type is declared to be private or limited private, then users of the package are restricted in their manipulations of any objects of the defined type. A separate package unit, called a body, contains the implementation of the specification. All information that is not essential to users (excluding the implementation of a private type) can be hidden in the body

The purchase of an Ada environment was made possible by a grant from the New Jersey Department of Higher Education Technical/Engineering Education Grant Program.

promoting the integrity of the encapsulation mechanism. Such packages can be readily made available to others, providing reusable software components. (The author placed packages in a directory available to students, who studied and used these packages for their programs.)

Pascal's failings in this area are clear. Separate compilation is not standard. It is difficult to physically group entities of data types, and to hide implementations. Type definitions and operations frequently clutter and obscure programs, as they are placed far from the code that calls them. (In how many students' Pascal programs have you requested that modules supplying operations for one type be grouped together?)

Generic packages and unconstrained array types are particularly effective in allowing generalizations of operations for similar structures. (Most texts introduce generics rather late in the material. The author's initial assignment was a generic package supplying operations such as square roots or exponentiation for a user defined float type. By the time stacks and linked lists were assigned, students used generics comfortably.)

The closest facility to generics in Pascal is supplied by variant records. Although uniform operations can be defined for structures with different component types, these types must be supplied at compilation time. Similarly, fixed bounds for each array type usually result in overly large amounts of storage allocated to objects, with the actual size recorded and passed as a parameter for array manipulation.

Names of operations and enumeration types can be overloaded in Ada. Thus infix operators and readable names can be used to denote operations and data values as they are commonly known. (Students coded real exponentiation operators, and complex, vector and matrix arithmetic operators, as well as arithmetic operators for their own NATURAL type.)

Ada's exception facility is particularly effective in handling error conditions with clarity and uniformity. Operations on abstract types, such as Integer division by 0, may fail. Operations may also fail because of limitations of an implementation, such as numeric overflow following Integer multiplication. The occurrence of system defined or user defined exceptions are assumed to be errors and therefore handled analogously at the end of operations, where they do not obscure the code. (For each data type implemented, students were required to define exceptions for error conditions. Exception handlers at the end of the operations continued to raise the exceptions, with the final handlers supplied by the package user. A driver program with several named blocks was used to test both normal and exceptional runs.)

Pascal, of course, has no such facility. Checking and handling all possible errors frequently clutters up more than half of a procedure's code. Nor does Pascal supply any structured way to distinguish between the handling of errors and the handling of boundary conditions.

A few students had programming backgrounds in C instead of Pascal. Again, the switch to Ada did not appear to be

difficult, with one student commenting that Ada forced him to program the way his instructors had taught him to program.

3. INSTANTIATION

The author considers packages, generics, and exception handling to be fundamental concepts necessary for the implementation of an abstract data type. These concepts were presented during the first few weeks, so that they could be continuously reinforced during the semester. Originally, their use required considerable correction. (One student used an exception handler to recursively call the containing module. "It works," he said.) By the middle of the semester, most students handled these constructs easily.

The author presented a generic template for the study of abstract data types, which was instantiated for both abstract data types and their implementations.

DATA TYPE

VALUES:

COMPONENT STRUCTURE: (default to void for scalar types)

OPERATIONS:

EXCEPTIONS: (default to void)

BOUNDARY CONDITIONS: (default to void)

Examples of instantiations included:

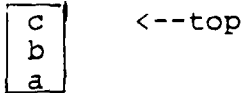
INTEGER of package STANDARD

VALUES: implementation dependent
-32768 .. 32767 for 16 bit 2's complement
representation

OPERATIONS: +, -, *, /, mod, rem, **, abs
relational operations
creation, destruction, copy
I/O available from TEXT_IO.INTEGER_IO
conversion to/from any numeric type
attribute qualification
additional operations supplied by users

EXCEPTIONS: NUMERIC_ERROR (division by 0, numeric overflow)
CONSTRAINT_ERROR (assignment of an out-of-range
value)

abstract type STACK

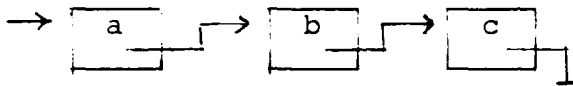
SAMPLE STRUCTURE WITH VALUES: 

OPERATIONS: creation, destruction
pop, push, test if empty

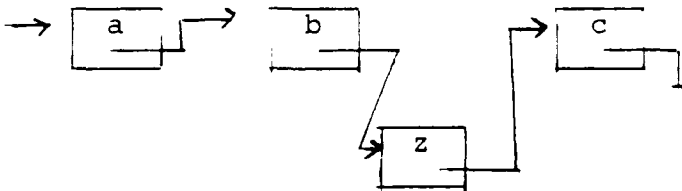
EXCEPTIONS: attempt to pop from an empty stack
attempt to push onto a full stack

LINEARLY LINKED LIST (programmer defined)

SAMPLE STRUCTURE WITH VALUES:



SAMPLE OPERATION:
INSERT after 2nd element



SAMPLE EXCEPTIONS:
Remove from an empty list.
STORAGE_ERROR is raised during INSERT if storage is exhausted (heap or array implementation)

SAMPLE BOUNDARY CONDITION:
INSERT, at the beginning of the list.

4. SUGGESTED ASSIGNMENTS

1) Code and instantiate a package that provides additional numeric operations on a programmer-defined numeric type. Exceptions are to be propagated by the package and handled by the calling routine. A sample operation would be real exponentiation, overloading the "**" operator. This operation requires handling those values that cause Overflow.

GOALS: Understanding the difference between an abstract data type and its implementation.

Introduction to generics.

Introduction to programmer-defined numeric types.

Introduction to exception handling.

Introduction to Ada.

2) Code and instantiate packages for stacks, queues or lists.

GOALS: Introduction to above types of data structures.

Introduction to information hiding, i.e.,

defining a type so that its components are either

a) completely available,

b) restricted in their use (with the private definition), or

c) completely hidden in the package.

Documentation, with stress on specifications.

3) Provide a dynamic storage allocation package using the stack package of example 2.

GOALS: Understanding dynamic storage allocation

Reusing software components previously defined.

4) Implement a LONG_NATURAL_NUMBER type providing arithmetic operations on long natural numbers, using the list package of assignment #2.

GOALS: Reusing software components

Numerics as composite not scalar types

Efficiency as an issue.

5) Implement a sort procedure using tasks.

GOAL: Introduction to tasking.

5. CONCLUSION

The author suggests that Ada be used at the sophomore level for a course in data structures, if not earlier. There are valid arguments for introductory material to be taught in Pascal, but the transition from Pascal to Ada is not difficult. Furthermore, Ada is clearly more effective for the presentation of the concept of an abstract data type, and for the secure, modifiable, and reusable implementation of this material.

Ada and Data Structures: "The Medium is the Message"

Melinda L. Moran
LCDR USN

NARDAC Washington
Washington, D.C. 20374
(202)475-7681

morani@ajpo.sei.cmu.edu

Introduction

Shakespeare did not HAVE to write in old English. Consider Romeo and Juliet in Latin. Rembrandt did not HAVE to paint in oils. Consider "Night Watch" in pastel water colours. A data structures course does not HAVE to be taught in Ada . . . but there is definitely "a message in the medium" a craftsman chooses.

Idealistically, a data structures course should be taught independent of any particular programming language. Realistically, some language must be chosen as the language of illustration by the instructor and the language of implementation by the student. Ada is not the *only* language which can be chosen. It is, however, one of the best, (if not *the* best), languages for illustrating and implementing the constructs central to a data structures course. These constructs must remain the focus of the course; the language must NOT predominate the focus but the language chosen can facilitate or frustrate the students' learning of these constructs. Ada most certainly facilitates.

Abstract Data Types

One of the most important constructs introduced in a data structures course is that of the "abstract data type (ADT)." The construct of an ADT

and the study of specific ADTs pervade the course. An abstract data type is the definition of a type of object and its associated values as well as the operations defined for objects of that type. Several ideas are central to teaching this construct. They are: 1) abstraction and modeling reality, 2) implementation, 3) information hiding, and 4) reliability.

Ada's package unit is ideally suited for illustrating and implementing abstract data types. The separation of package specification and package body clearly parallels the separation of abstraction and implementation students are being taught. Exercises can be easily created by the instructor where the student is given the abstraction (specification) for an ADT and is required to substitute a different implementation (body) and observe the minimal effects on program units which make use of the ADT. The exercise used by this instructor was that of having students change the implementation of a queue used as a supporting data structure in a larger project of simulating an airport. (Details of this project are included at the end of this paper.)

Further, the separation of specification and body allows the student to focus first on the functionality (i.e. the abstraction) of a new ADT before becoming immured and overwhelmed with the implementation details. The student can see, in practice, the utility of information hiding. (S)he is not forced to immediately focus on the implementation details which are carefully hidden away in the body of the package. (S)he first *uses* the specification to access the services of the package before becoming concerned with *how* these services are provided.

The exercise used by this instructor to illustrate the constructs of information hiding and levels of abstraction was that of implementing a "long integers" package, where a long integer was defined to be a non-negative integer with an unlimited number of digits. Students were given a generic stack package and instructed in how to instantiate to create a stack package for integers. They then used this package to create their long integers package. In being *given* the stack package and in learning how to use its specification to access the services of the package, students were able to study the abstraction of a stack and learn about its utility as a data structure in helping them to solve the higher level or layer of the problem they were working on, that of creating a long

integers package. The second step was to have students focus on the lower level of the problem, that of analyzing (and perhaps changing) the implementation of the stack ADT. (Details of the long integer project are included at the end of this paper.)

In conjunction with Ada's packaging feature, the ability to create user-defined types and to define subtypes greatly facilitates the ability to model reality when creating a package to embody an ADT. Being able to create user-defined types and define subtypes allows the imposition of constraints upon objects of those types, constraints which clearly mirror the real world objects which they model. A favorite exercise of this instructor in teaching the construct of modeling reality was to bring a collection of familiar objects (such as cans of various types of soda pop) to class and guide students in creating the specification of a package to embody the ADT representing that collection of objects. Students can readily see how the use of user-defined types and subtypes can make a program more readable/understandable if it closely models the real world. (Two such sample specifications are included at the end of this paper.)

Ada's exception handling feature plays a very supportive role in teaching students to create ADTs which are reliable. By the time they reach a data structures course, most students have already completed an introductory computer science course and are well acquainted with the reality of fatal programming errors. They realize the elaborate checking that must be built into the code of programs written in languages such as Pascal and BASIC to attempt to capture these errors, and, they realize that certain errors are simply impossible to catch. Ada's exception handling feature offers a welcome relief, a relatively painless way to plan for and capture fatal errors.

Ada's tasking facility further supports teaching students to develop ADTs which are reliable. Ada is one of very few languages with the construct for concurrency built into the syntax and semantics of the language itself, yet, concurrency is an issue which certainly must be considered in developing a reliable ADT. More and more our students, perhaps more so than ourselves, will have to deal with the issues of concurrency as architectures underlying their software become increasingly parallel. In Ada, it is a simple exercise to have students

create a program with multiple tasks all accessing the same data structure. And, again, because of the separation of abstraction and implementation, it is easy for the student to change the body of a stack package, for example, and observe what happens when multiple tasks access a "sequential" stack as opposed to a "concurrent" stack.

Reuse and Building Software Systems

Moving beyond the construct of ADTs, Ada affords the data structures instructor significant support in introducing another construct which, in this instructor's opinion, should certainly be a central part of any data structures course. This is the construct of software reuse.

Students of modern data structures courses must be taught to "build" software systems from existing units, not to "code" them from the ground up. The increasing demand for software today and the lack of productivity in the latter approach makes it infeasible. Further, the increasing intelligence found in modern compilers obviates the need for concern over the inefficiencies introduced in reusing software units. Good compilers are quite capable of optimizing away unused resources in a reusable unit.

The introduction of procedures and functions could be viewed as one of the first steps in the evolution of software reuse. Procedures and functions are the creation of reusable modules within a program. Ada's package and Modula 2's module could be viewed as the next step up this evolutionary chain of reuse. Both allow the encapsulation of related resources for reuse within other units of a software system. Ada's generics feature is yet another step up the chain. Generic packages and subprograms allow the introduction of abstraction at yet a higher level. They provide the facility for creating potentially parameterized templates which extract the essence of multiple specific instances of packages and subprograms.

Libraries of generic units which embody the various ADTs and algorithms discussed in data structures courses are becoming increasingly available commercially and easy to acquire at reasonable prices. Their use can facilitate tremendously the amount which students can learn in a data structures course.

One way in which the use of libraries of generic units can facilitate student learning was mentioned earlier. By instantiating a generic unit and giving a student access to the specification (but not the body) of a particular ADT or algorithm, experience can be gained by the student in the functionality of the unit. The student, for example, can be given easy access to multiple sorting algorithms in the library. (S)he can "play" with these algorithms as applied to a variety of data sets and get a sort of "intuitive feel" for the circumstances under which each algorithm performs optimally. After this top level functionality has been studied, the student can then either be given the implementation details in the unit body for analysis and, perhaps, modification or be asked to synthesize that unit body based on his earlier experience. The student is not so caught in panic over the low level details of implementation that (s)he fails to glean any functional knowledge about the performance characteristics of the algorithm. Both levels of knowledge must be acquired in a data structures course but with undergraduates it is a very real concern that "implementation anxiety" often impedes other learning.

The second way in which the use of libraries of generic units can facilitate student learning is also by reducing the focus on coding and freeing the student to focus on higher level concerns in building a system such as the efficiency of choosing a particular data structure or algorithm. With the availability of libraries of generic units, it is feasible for instructors to require larger projects which are much more like real software systems the student will encounter after graduation. Without the provision of such libraries, instructors in the past have been constrained simply by available time as to the size of a project students could reasonably be expected to complete. And students, in the past, have been so focused on simply coding fast enough to finish a project that they have not always had the flexibility to do any comparative analysis of different implementations. The exercise used by this instructor in teaching students to consider the efficiencies of choosing data structures in a large system was to have students create a system to simulate the software in a video rental store. (Details of this project are included at the end of this paper.)

A final way in which the support provided by libraries of generic units can facilitate student learning in a data structures course is in the use of generic units representing certain ADTs to create other generic

units representing related ADTs. Two specific examples of this are 1) the creation of a generic package representing the stack ADT and 2) the creation of a generic package representing the queue ADT both from a generic package representing the list ADT. It is always pointed out that the stack ADT and the queue ADT are simply the list ADT with more stringent constraints applied. Using a generic library unit for a list ADT and the renames clause it is simplistic to create these new constrained ADTs. Students can very clearly see the relationships and learn to quickly create new ADTs by constraining existing ones. (Specific examples of these "derived" ADTs are included at the end of this paper.)

Conclusion

Ada is not the only language suitable for illustration and implementation of constructs in a data structures course, but the facilities Ada provides certainly make it one of the best languages currently available. The features of packaging, separate compilation of specification and body, user-defined types, exception handling, and concurrency support the instructor's illustration of and facilitate the students' understanding and implementation of ADTs. The feature of generics supports the illustration and implementation of software reuse and allows both the data structures instructor and student to shift their focus from low-level coding concerns to higher level analysis of ADTs and algorithms. In short, Ada as a medium for illustrating and implementing data structures constructs facilitates both effective teaching and learning of these constructs.

Si301 Data Structures
Project #1

I. Problem Statement: You need to add or multiply two HUGE non-negative integer numbers. Let's call these numbers "long integers." Long integers may contain an unlimited number of digits.

Example:

```
984576541479974245654378945563894329865321277421985375429876547  
X 890808734521879539987025436780932876087769211394985543766885418  
-----
```

(There is no particular pattern or meaning to the numbers in the example. They were generated arbitrarily as examples.)

II. Assignment - Part 1 - due beginning of class 29 AUG 1988

- A. Requirements Analysis - Read the problem carefully and answer the following questions.
1. Exactly WHAT is the task defined by the problem?
 2. What are the constraints in the problem?
 3. What other questions would you like to ask about the problem?
 4. What alternatives exist in solving the problem? (Be sure to consider "non-computer" solutions.) Briefly describe why each considered solution is or is not viable (achievable).
 5. For a "computer" solution, what resources are available that might be useful and where might you look for them?

III. Assignment - Part 2 - due beginning of class 31 AUG 1988

- A. Design - It is determined that a "computer" solution best fits the problem. You are going to create the package defined below. BEFORE you leap into the "code and pray" mode, answer the following questions.
1. What data structure are you going to use to represent a long integer object?
 2. Describe in simple English phrases how you plan to implement each requirement defined for the package. Draw pictures if it will help you explain but do NOT WRITE CODE!! (Be cautious! Don't let your partner pressure you here. (S)he may be suffering from WAYWACY (Why Aren't You Writing Any Code Yet). DESIGN IS IMPORTANT...MORE THAN CODING!!
 3. For each requirement defined in the package list the exceptions you will need to plan for in that routine and what action you will take for each. (i.e. What errors is it possible might occur in your algorithm or might be introduced by the user of your package?)

IV. Assignment - Part 3 - due on computer by 0800 07 SEP 1988

Code and compile the SPECIFICATION for a package called Long_Integers which allows a user to do the following:

- 1) declare multiple objects of the type Long_Integer
- 2) assign a value to an object of type Long_Integer
- 3) add two objects of type Long_Integer using infix notation (i.e. $L3 := L1 + L2;$)
- 4) multiply an object of type Long_Integer by a natural number (0, 1, 2, ...) using infix notation (i.e. $L2 := N * L1;$)
- 5) multiply two objects of type Long_Integer using infix notation (i.e. $L3 := L1 * L2;$)
- 6) print a long integer object on the screen

The specification must be in a file called XXXXXXXXXX located somewhere in your directory by 0800 on the due date.

V. Assignment - Part 4 - due on computer by 0800 09 SEP 1988

Code and compile the BODY for the Long_Integers package and then code, compile, link, and execute a test procedure to test the correctness of your Long_Integers package.

The body must be in a file called Long_Integer.ada located somewhere in your directory by 0800 on the due date. Your test procedure will not be graded for this project.

Si301 Data Structures
Project #2

I. Problem Statement:

You want to simulate an airport landing and takeoff pattern. The airport has 3 runways, runway 1, runway 2, and runway 3. There are two landing holding patterns for each runway. As a plane arrives it must enter at the end one of these landing holding patterns. All landing holding patterns should be kept as equal in size as is possible.

Each runway also has one takeoff pattern. Each plane desiring to takeoff must enter at the end one of these takeoff patterns. All three takeoff patterns should be kept as equal in size as is possible.

Planes entering the landing holding patterns each possess an integer ID number and an integer indicating how many fuel units (same as flying time units) they have left. This value decreases by one at each time interval in the simulation.

At each time interval 0 - 3 planes may arrive at the airport ready to enter the landing holding patterns and 0 - 3 planes may be ready to enter the takeoff patterns. Each runway can handle one takeoff or one landing at each time interval. Planes in the landing holding patterns whose remaining fuel has reached zero must be given priority to land over other planes taking off or landing. Lengths of landing and takeoff patterns should not be allowed to grow excessively.

Neither landing nor takeoff patterns may be sorted. Planes must be entered into the patterns AT THE END of the existing pattern. With the exception of zero fuel emergencies planes entering the pattern first must leave the pattern first.

Input to the simulation will be from a text file. There will be a pair of lines for each time interval in the simulation. The first line will contain the number of planes desiring to enter the landing holding patterns followed by the appropriate number of pairs of id number - remaining fuel unit pairs. The second line will contain the number of planes desiring to enter the takeoff patterns followed by the appropriate number of id numbers.

Example:

```
3 197 45 200 30 458 14
2 187 479
```

There are 3 planes desiring to land. Their ID numbers are 197, 200, and 458. Plane 197 has 45 remaining fuel units; plane 200 has 30 remaining fuel units and plane 458 has 14 remaining fuel units. There are 2 planes desiring to takeoff. Their ID numbers are 187 and 479.

Required output from the simulation is a report at each time interval indicating which planes landed (and what their remaining fuel units were) and which planes took off. At each fifth time interval a summary report must also be produced indicating the average takeoff waiting time for all planes that have taken off so far, the average remaining fuel units for all planes that have landed so far, the number of emergency landings so far, the average waiting time for all planes that have landed so far, and the current contents of each pattern in terms of first plane's ID number to last plane's ID number. Output should be well labeled and organized into an easily read report.

II. Assignment - Part 1 - due beginning of class 12 SEP 1988

- A. Requirements Analysis - Read the problem carefully and answer the following questions.
1. Exactly WHAT is the task defined by the problem?
 2. What are the constraints in the problem?
 3. What other questions would you like to ask about the problem?
 4. What alternatives exist in solving the problem?
 5. For a "computer" solution, what resources are available that might be useful and where might you look for them?

III. Assignment - Part 2 - due beginning of class 14 SEP 1988

- A. Design - It is determined that a "computer" solution best fits the problem. BEFORE you leap into "code and pray" mode, answer the following questions.
1. What are the objects defined in the problem?
 2. What are the operations that will be performed by and on each object?
 3. What data structure are you going to use to represent a landing or takeoff pattern?
 4. What data structure are you going to use to represent a plane?
 5. What packages are you going to need in your system?
 6. What is your report going to look like? How will it be formatted?
 7. For each package, describe in simple English phrases the objects and routines that will be included. Draw pictures if it will help but do NOT WRITE CODE!
 8. For each routine in a package list the exceptions you will need to plan for in that routine and what action you plan to take for each.

IV. Assignment - Part 3 - due on computer by 0800 19 SEP 1988

Code and compile the specification(s) for the package(s) you determined you will need. Additionally, using the editor, create a file called FILENAMES.DAT somewhere under your account. In this file list the name of each package

spec you created. The name of the file that a spec is located in must be the name of the package plus a trailing underscore. For example, if you create a package spec for a package called DOODLE then that spec must be located in a file called DOODLE_.ada located somewhere under your account.

Code and compile the main procedure that will drive your simulation. The main procedure must be called BWI and must be placed in a file called BWI.ada located somewhere under your account.

You obviously cannot link or execute your main procedure yet because the bodies of your packages have not been completed.

V. Assignment - Part 4 - due on computer by 0800 23 SEP 1988

Code and compile the bodies for your package(s). Code, compile, link, and execute the main procedure for your system. Test until you are convinced it works correctly. You will need to use the editor to create a file of test input data as defined in the problem statement. This test data must be placed in a file called BWI.dat located somewhere under your account. Your package bodies must be located in files whose names are the same as those of the specs but without the trailing underscore. For example, the body for package DOODLE would be in a file called DOODLE.ada located somewhere under your account.

VI. Assignment - Part 5 - due on computer by 0800 30 SEP 1988

Change the underlying implementation of the queue package from that of a singly linked linear list to that of a singly linked circular list. The specification for the queue package will remain the same. A copy is attached. You will create a new package body with the same routines but where the list implementing the queue structure is circularly linked.

Code and compile and test the new package body. Do NOT recompile any other component of your airport simulation system.

Relink and rerun your main procedure. No visible changes should occur between the original and this changed version since the implementation of the queue should not affect the upper levels of abstraction in your system.

SI301 Data Structures
Project #3

I. Problem Statement: It is the year 1990. Every midshipman now has a VCR installed in their room. The midshipman store has decided to go into the video tape rental business. You are an Ensign assigned to help the midshipman store develop the software system necessary to manage the midshipman accounts and the inventory of tapes.

Specifically the system must:

- A) load the initial inventory of movies contained in the text file located at `sys$courses:[si301]movies.dat` into a binary search tree ordered on movie title.
- B) load the initial inventory of midshipmen members contained in the text file located at `sys$courses:[si301]mids.dat` into a data structure of your choice.
- C) allow the addition of new movies to the inventory. Information about each movie stored will include (at minimum) its title, its rating (General (G), Parental Guidance (PG), Restricted (R), Non-Rated (NR)), its length in minutes, its category (Classic (CL), Comedy (CO), Drama (DR), Mystery (MY), Musical (MU), Adventure (AD), Science Fiction (SF)), its year of release, and its unique bar code number.
- D) allow the addition of new members to the club. Information about each member will include their name, a unique card number (a natural number between 1 and natural last generated by your system), their company, and major.
- E) allow a member to check out and return movies. There is no limit on the number of movies a member may check out. There is only one tape of each movie available, however, so a member cannot check out a movie which has previously been checked out by another member.
- F) be capable of producing the following on-screen reports:
 - 1) a report of which movies are currently checked out by a SPECIFIC member. Report to include all info on each movie.
 - 2) a report of all movies currently checked out listed in order from most recently checked out down to least recently checked out.

II. Assignment - Part 1 - due beginning of class Monday,
17 OCT 1988

- A. Requirements Analysis - Read the problem carefully and answer the following questions.
 - 1. Exactly WHAT is the task defined by the problem?
 - 2. What are the constraints in the problem?

3. What other questions would you like to ask about the problem?
4. What alternatives exist in solving the problem? (Be sure to consider "non-computer" solutions.) Briefly describe why each considered solution is or is not viable (achievable).
5. For a "computer" solution, what resources are available that might be useful and where might you look for them?

III. Assignment - Part 2 - due beginning of class 21 OCT 1988

- A. Design - It is determined that a "computer" solution best fits the problem. You are going to create the package defined below. BEFORE you leap into the "code and pray" mode, answer the following questions.
1. What objects can you identify in the problem?
 2. What operations are performed by and/or suffered by each object?
 3. What data structure are you going to use to represent each object?
 4. What packages are you going to need in your system?
 5. Describe in simple English phrases how you plan to satisfy each requirement defined in the problem. Draw pictures if it will help you explain but do NOT WRITE CODE!! (Be cautious! Don't let your partner pressure you here. (S)he may be suffering from WAYWACY (Why Aren't You Writing Any Code Yet). DESIGN IS IMPORTANT...MORE THAN CODING!!
 6. For each requirement defined in the problem list the exceptions you will need to plan for in the routine that will address that requirement and what action you will take for each. (i.e. What errors is it possible might occur in your algorithm or might be introduced by the user of your package?)

IV. Assignment - Part 3 - due on computer by 0800 31 OCT 1988

Code and compile the specification(s) for the package(s) you determined you will need. Additionally, using the editor, create a file called FILENAMES.DAT somewhere under your account. In this file list the name of each package spec you created. The name of the file that a spec is located in must be the name of the package plus a trailing underscore. For example, if you create a package spec for a package called DOODLE then that spec must be located in a file called DOODLE_ada located somewhere under your account.

Code and compile the main procedure that will drive your system. This procedure must be called EROLS and must be in a file called EROLS.ada located somewhere under your account. Your program must consist of a loop which contains a menu of choices consisting of items C-F from Part I.

You obviously cannot link or execute your main procedure yet because the bodies of your packages have not been completed.

Your objective is to get as clean and modular a design as possible and choose as efficient data structures as you can to optimize performance of your program. You may use any package you have been given so far and you may request a copy of any package mentioned in the Booch Software Components with Ada. To request a copy of a package merely send mail with its name to your instructor.

V. Assignment - Part 4 - due on computer by 0800 14 NOV 1988

Code and compile the bodies for your package(s). Code, compile, link, and execute the main procedure for your system. Program bodies must be placed in files whose names are the same as the file names of their corresponding specs except the final underscore will NOT be present. Test until you are convinced it works correctly. You will need to use the editor to create a file of test input data as defined in the problem statement.

```

-- Module:   Soft Drinks
-- Author:   LCDR MORAN
-- Date:    14 SEP 1987
-- Function: Allows the user to declare and operate on variables of type
--           SODA (i.e. a soft drink model.)

```

```

package SoftDrinks is

```

```

    type FlavourType is (cherry, grape, orange, black_cherry, cola,
                          chocolate_fudge, none);
    type BrandType is (Shasta, Canfields, SevenUp, CocaCola, none);
    type Ounces is range 0 .. 20;
    type PriceType is digits 2 range 0.0 .. 0.75;
    type ContainerType is (bottle, can, none);

```

```

    type Soda is

```

```

    record

```

```

        Flavour      : FlavourType      := none;
        Brand        : BrandType        := none;
        Diet         : boolean          := false;
        Container    : ContainerType     := none;
        ContainerSize : Ounces           := 0;
        ContentsRemaining: Ounces       := 0;
        Open         : boolean          := false;
        Price        : PriceType        := 0.0;

```

```

    end record;

```

```

    procedure OpenSoda(TheSoda : in out Soda);
    -- used to indicate TheSoda has been opened

```

```

    procedure DrinkSoda(TheSoda : in out Soda; GulpSize : in Ounces);
    -- decreases contents remaining in TheSoda by GulpSize

```

```

    procedure ThrowAwaySoda(TheSoda : in out Soda);
    -- used to indicate TheSoda has been disposed of

```

```

    function Flavour(TheSoda : Soda) return FlavourType;
    -- return the flavour of the soda (cherry, grape, orange, etc.)

```

```

    function Brand(TheSoda : Soda) return BrandType;
    -- return the brand of the soda (Shasta, CocaCola, etc.)

```

```

    function IsDiet(TheSoda : Soda) return boolean;
    -- return TRUE if the soda is a diet soda, FALSE otherwise

```

```

    function ContainerSize(TheSoda : Soda) return Ounces;
    -- return the number of ounces in the soda

```

```

    function Container(TheSoda : Soda) return ContainerType;
    -- return the type of container the soda is in (i.e. bottle, can, etc.)

```

```

    function Price(TheSoda : Soda) return PriceType;
    -- return the price of the soda

```



```
function ContentsRemaining(TheSoda : Soda) return Ounces;  
-- return the number of ounces remaining in the soda  
  
function IsOpen(TheSoda : Soda) return boolean;  
-- return TRUE if the soda is open, FALSE otherwise  
  
end SoftDrinks;
```

```
package Cars is
```

```
type Make is (Porsche, Honda, Mercedes_Benz, Rolls_Royce);  
subtype Year is integer range 1950..2000;  
subtype Price is float range 0.0..999999.99;  
type Colour is (white, black, red, silver, blue, green);  
subtype NumOfCylinders is integer range 4..10;
```

```
type Car is private;
```

```
function CreateCarObject(Description : in Make;  
                        ModelYear   : in Year;  
                        Cost        : in Price;  
                        PaintJob    : in Colour;  
                        Cylinders   : in NumOfCylinders;  
                        Stereo      : in boolean)  
return Car;
```

```
function GetPrice(TheCar : in Car) return Price;
```

```
function GetDescription(TheCar : in Car) return Make;
```

```
-- other GET functions for each field of the car object --
```

```
procedure Repaint (TheCar : in out CAR; NewPaint : in Colour);
```

```
procedure UpThePrice(TheCar : in out CAR; PriceIncrease : in Price)
```

```
procedure LowerThePrice(TheCar : in out CAR; PriceDrop : in Price)
```

```
procedure InstallStereo(TheCar : in out CAR);
```

```
procedure StereoStolen(TheCar : in out CAR);
```

```
-- other procedures to update fields of the car object --
```

```
private
```

```
type CAR is record
```

```
    Descr      : Make           := Porsche;  
    YrMade     : Year           := 1989;  
    Cost       : Price         := 150000.00;  
    Stereo     : boolean       := true;  
    PaintJob   : Colour        := silver;  
    Cylinders  : NumOfCylinders := 6;  
end record;
```

```
end Cars;
```

```

generic
  type Item is private;
package List_Single_Unbounded_Unmanaged is

  type List is private;

  Null_List : constant List;

  procedure Copy      (From_The_List : in List;
                      To_The_List  : in out List);
  procedure Clear    (The_List      : in out List);
  procedure Construct (The_Item      : in Item;
                      And_The_List  : in out List);
  procedure Set_Head (Of_The_List   : in out List;
                      To_The_Item   : in Item);
  procedure Swap_Tail (Of_The_List  : in out List;
                      And_The_List  : in out List);

  function Is_Equal (Left      : in List;
                    Right     : in List) return Boolean;
  function Length_Of (The_List : in List) return Natural;
  function Is_Null   (The_List : in List) return Boolean;
  function Head_Of   (The_List : in List) return Item;
  function Tail_Of   (The_List : in List) return List;

  Overflow      : exception;
  List_Is_Null : exception;

private
  type Node;
  type List is access Node;
  Null_List : constant List := null;
end List_Single_Unbounded_Unmanaged;

```

*[*from Software Components with Ada by Gregory Jochims]*

```

with List_Single_Unbounded_Unmanaged;
generic
  type Item is private;
package Stacks is

  package LIZT is new List_Single_Unbounded_Unmanaged(item=>item);

  subtype Stack is LIZT.List;

  procedure Copy (From_The_Stack : in Stack;
                 To_The_Stack   : in out Stack)
    RENAMES LIZT.COPY;

  procedure Clear (The_Stack      : in out Stack)
    RENAMES LIZT.CLEAR;

  procedure Push (The_Item       : in Item;
                 On_The_Stack   : in out Stack)
    RENAMES LIZT.CONSTRUCT;

  procedure Pop (The_Stack       : in out Stack);

  function Is_Equal (Left        : in Stack;
                    Right       : in Stack) return Boolean
    RENAMES LIZT.IS_EQUAL;

  function Depth_Of (The_Stack : in Stack) return Natural
    RENAMES LIZT.LENGTH_OF;

  function Is_Empty (The_Stack : in Stack) return Boolean
    RENAMES LIZT.IS_NULL;

  function Top_Of (The_Stack : in Stack) return Item
    RENAMES LIZT.HEAD_OF;

  Overflow : Exception RENAMES LIZT.OVERFLOW;
  Underflow : Exception RENAMES LIZT.LIST_IS_NULL;

end Stacks;

```

```

package body Stacks is

  procedure Pop (The_Stack : in out Stack) is
  begin
    if Is_Empty(The_Stack) then
      raise Underflow;
    else
      The_Stack := LIZT.TAIL_OF(The_Stack);
    end if;
  end Pop;

end Stacks;

```

```

-- Module      : Lists
-- Author     : LCDR MORAN
-- Date      : 29 SEP 1987
-- Function   : Implements basic operations on a singly linked list.

```

```
generic
```

```

type Item is private;
type KeyType is private;

with function Key(AnItem : Item) return KeyType;
with function LE(Key1, Key2 : KeyType) return boolean;
with function EQ(Key1, Key2 : KeyType) return boolean;
package Lists is

    subtype Count is natural;

    type ListPointer is private;

    procedure Copy(PointerToOriginalList : in ListPointer;
                   PointerToCopyList   : out ListPointer);

    procedure Clear(PointerToTheList : in out ListPointer);

    procedure Share(PointerToOriginalList,
                    PointerToSharingList : in out ListPointer);

    procedure InsertAtHeadOfList(PointerToTheList      : in out ListPointer;
                                  TheItemToBeInserted : in Item);

    procedure InsertAtTailOfList(PointerToTheList      : in out ListPointer;
                                   TheItemToBeInserted : in Item);

    procedure InsertInOrderInList(PointerToTheList      : in out ListPointer;
                                    TheItemToBeInserted : in Item);

    procedure RemoveFromHeadOfList(PointerToTheList      : in out ListPointer;
                                      RemovedItem        : out Item);

    procedure RemoveFromTailOfList(PointerToTheList      : in out ListPointer;
                                      RemovedItem        : out Item);

    procedure RemoveByKeyFromList(PointerToTheList      : in out ListPointer;
                                    RemovedItem        : out Item;
                                    KeyValue           : in KeyType);

    function AreEqual(PointerToL1, PointerToL2 : ListPointer) return boolean;

    function IsEmpty(PointerToL : ListPointer) return boolean;

    function LengthOf(PointerToL : ListPointer) return Count;

    function Predecessor(PointerToAList, PointerToANode : ListPointer)
        return ListPointer;

    function Successor(PointerToAList, PointerToANode : ListPointer)
        return ListPointer;

    function GetData(PointerToANode : ListPointer) return Item;

    EmptyList : exception;

```

```
private
  type ListNode;
  type ListPointer is access ListNode;
end Lists;
```

```

with Lists;
generic
  type Item is private;
package Queues is

  function Key(AnItem : Item) return Item;

  function LE(Key1, Key2 : Item) return boolean;
  function EQ(Key1, Key2 : Item) return boolean;

  package Q is new Lists(Item=>Item, KeyType=>Item,
    Key=>Key, LE=>LE, EQ=>EQ);

  subtype Queue is Q.ListPointer;

  procedure Copy(Q1 : in Queue; Q2 : out Queue)
    RENAMES Q.COPY;

  procedure Clear(Q1 : in out Queue)
    RENAMES Q.CLEAR;

  procedure Insert(Q1 : in out Queue; AnItem : in Item)
    RENAMES Q.INSERTATTAILOFLIST;

  procedure Remove(Q1 : in out Queue; AnItem : out Item)
    RENAMES Q.REMOVEFROMHEADOFLIST;

  UNDERFLOW : Exception RENAMES Q.EMPTYLIST;

end Queues;

```

A Healthy Marriage : Ada and Data Structures

Harold Youtzy, Jr.
Department of Mathematics and Computer Science
Briar Cliff College
Sioux City, Iowa 51104

Introduction. In an earlier paper [1], I proposed introducing Ada after the first and second course in computer science. Since then, I have followed that advice and have twice taught Data Structures using Ada. The first experience yielded results far inferior to what was hoped for. However, the second time around has borne the type of results from which encouragement is derived! In this paper, I plan to reflect on those two experiences, the changes that were made, and the resulting effects upon our curriculum.

The Setting. Briar Cliff College is a small, Catholic, liberal arts college with an enrollment of 1200 students, two thirds of whom are full time. The computer science major was introduced in 1985. Following the national trend, the number of majors has yet to reach what was anticipated, although it has been steady.

Chapter One. With high expectations following our initial contact with Ada, Data Structures was taught using Ada for the first time in 1987/88. Pascal is the primary language in the CS1 and CS2 courses, and had always been used in the Data Structures course as well. Introducing Ada here was thus the first look at the language by the students. Unfortunately, many left hoping it would also be their last!

Numerous pitfalls led to the failure of that first attempt. My own lack of adequate preparation, and failure to anticipate the types of problems (and complaints) that would arise from the students admittedly played a major role. It is not in vogue to acknowledge our part in failure, and though it may be easier to pawn it off on other contributing factors, I must stand up to face my share of the music for its demise. Having taught Ada as a programming language in the previous year, I had uncovered many of the organizational problems of introducing a new language. As such I felt adequately prepared to use Ada within the Data Structures course. But teaching it as a language and using it as a tool for data structures are two different approaches, and one approach does not necessarily prepare you for the second. For example when teaching it as a language, a Pascal-like procedural approach was followed in writing the programs, but when using it as a tool for data structures, the emphasis was placed on defining the data structure and its operations within packages.

Additionally, the set of available textbooks is limited. This gave way to two deficiencies. First, Booch's text, Software Components With Ada [2], was selected, in part, due to the large amount of code that was included in the text. The preface of the text highlights Data Structures as one of its intended uses and presumes a basic understanding in a high-order language such as Pascal. With no prior exposure to Ada, it seemed reasonable that an abundance of code would assist the students in grasping the syntax of the language. And given its flavor for reusable software components, the book seemed to be a perfect match. Unfortunately, it seems that compilation errors play a far more memorable role in teaching syntax than does cursory examination of code in a textbook. The students were able to grasp the "big picture" but grew weary of crossing their t's and dotting their i's. It was also apparent that "deskchecking" has been relegated to an interactive activity. They had become accustomed to quick compilation with Pascal, and as fast as the Pascal compiler would flag an error, they could correct it. It seemed inappropriate to check your programs beforehand when the compiler could do it for you. As such, the students unwittingly contributed to their slow compilation speed by having a substantial number of syntax errors in their programs. At times, it appeared that the compilation speed was exponentially proportional to the number of syntax errors within the program.

Second, with regard to textbook selection, the principal theory behind the individual data structures was not as pronounced as in other books devoted strictly to data structures. This factor, combined with my aforementioned failure, left woefully inadequate the repetition that is sorely needed by the students when learning new principles. It thus appeared to the students that theory had been traded for code.

Observing the difficulties confronting the students my attempt to solve the problem, or at least ease their discomfort, only served to fuel the fire that had by now reached at least two alarm status! Instead of providing them with the additional repetition, more time was spent studying the language itself. Since the language had now become their major source of frustration, sessions devoted to teaching the essential elements of Ada often became opportunities for the students to berate the language. All of this, was, of course, distracting us from the objectives of the course. In retrospect, I may have been able to salvage the class at this point had I provided more theory and a closer examination of the reasoning behind the development of the code found in Booch's text. Unfortunately, neither action was taken, nor was the class salvaged.

Lastly, the steamroller effect was also a participant. Almost everyone entered the course enthusiastically. The prospect of acquiring a feel for Ada in addition to learning the normal data structures material was considered a plus. In terms of academic ability, they were unquestioningly one of the better classes to enter the course. However, when they began to

recognize that their own natural high expectations for the course were not going to be realized, their hopes diminished as did their excitement about becoming familiar with the language. As the morale of the troops deteriorated, so too did the morale of their leader. The checkered flag, also known as the end of the course, waved vigorously in the thoughts of all involved as we sped to our disastrous conclusion.

Chapter Two. Having learned the lessons from the previous year all too well, a second adventure into using Ada in data structures began with much more thought and preparation. With the stories of Ada filtering their way (or should I say ballooning their way?) to the next class of students, there was no doubt more trepidation than excitement in most of their minds.

As a starter, new books were introduced to the class. Feldman's text, Data Structures with Ada [3] was selected as a required text. Ada as a Second Language [4], by Cohen, was adopted as a recommended text. Although a simple change in texts rarely produces a resounding difference, it stands as one of the most influential factors in the outcome of the course. Feldman does a good job of combining theory with practice. An ample supply of examples written in Ada provided the students with the chance to view the theoretical principles in coded form. I also made certain to underscore these principles as we progressed through the book. How unfortunate that so much of our learning has to take place through failure!

The addition of Cohen's text also made an impact. In the previous year, questions regarding the language itself had to be answered from the language manual. Although that idea is sound, we all know how difficult it is to search any language manual, let alone Ada's, to answer a question about the language. So in our current year, questions of syntax and semantics were relegated to Cohen's text. Its 800+ pages provide discussion for all the topics addressed by students in this class regarding the language. And though it was a recommended text, all the students soon recognized its value, and none went through the course without it. The only complaint voiced by the students with regard to Cohen's text was the fact that he introduces a BasicIO package at the beginning of the text which he uses throughout. The students felt it would be better to stick with the predefined input output packages provided by Ada. Nonetheless, Cohen's text was a more beneficial reference tool than was the standard language reference manual.

A second major factor was influenced somewhat by what was just mentioned. In my first attempt to teach the course, a disproportionate amount of time was spent on the language. It was very easy to get caught up in teaching Ada as a language instead of a tool to implement the structures being studied. In my second approach, I paid careful attention to provide instruction to the students with only that portion of the

language necessary to successfully implement the details of the theory into their programming assignments, homework, and daily reading. As such, the course stayed within its objectives at all times. In fact, rather than lagging behind in the syllabus, we completed Feldman's book with time still remaining at the end of the course. At no time in the course did I feel that we were rushing through the text. The coverage of the principles and theories seemed well balanced, and though the academic ability of the second class was not exceptional, as was the first, most of the students came away with a greater comprehension of the course material.

Thirdly, given the horror stories provided by the previous class, the second class of students were more earnest in doing "deskchecking." A VAX 11/750 services the needs of the students in most of the programming courses, and although the VAX is overworked at times, turn-around time for the students never reached the epidemic level it had in the previous year. Once or twice in the prior year, an overnight turn-around time was experienced. In the present year, seldom did it go beyond a half hour. (All programs were run via batch mode.) Students, on their own, brought hand-written code in to be screened before submitting it to the compiler! And when they experienced compilation errors, they often went first to Cohen's text to resolve their errors. Performance for the TeleSoft Ada compiler has improved over the past year, but its improvement paled in comparison to the students' improved performance in screening the programs before submitting their code. Hence, the students recognized the value of deskchecking their programming assignments, and were benefitted, in return, by much faster compilation turn-around time.

Additionally, just as the steamroller effect can be negative, it can, in like manner, be positive. The students' trepidation at the beginning of class slowly faded as they saw the changes in outcome from the previous year. A clearer perspective, and a year of experience, allowed me to be better prepared for the problems sure to arise from the students. They more easily saw the benefits of using Ada rather than Pascal. Although to some degree this was because they didn't face the problems with Ada that was true of the previous class, I would also like to believe that by unveiling only portions of the language, they never got caught up in the immensity of Ada. I believe students in the prior class sometimes felt overwhelmed by its size, to the degree that perhaps they felt it insurmountable. Without taking time to fix leaks in the dam, more time was spent doing repetition, giving students a better understanding of how to arrive at the necessary solutions. The fact that they were more comfortable with their understanding of the material contributed immeasurably to the steamroller effect. As such, their leader was better able to cry "charge" rather than "retreat!"

Lastly, also related to what was just mentioned, some improvement was made because my own expectations of the academic ability of the class were not as great. In the first class I recognized their excellent academic ability and left much of the teaching on their shoulders. Instead, I should have increased the level of my own teaching. The second time around I paid stricter attention to what I was teaching and how I was teaching, making sure that I was communicating effectively with all the students in the class.

Chapter Three. Where do we go from here? Many would assume the next step would be to incorporate Ada as the primary language. However, we remain firmly committed to Pascal, especially with regard to CS1. "Safety in numbers" is an easy excuse to keep Pascal as the primary language. The vast majority of colleges and universities continue to use Pascal as their principal language. But for us, the availability of resources serves as the major stumbling block. As resources for Ada become increasingly accessible, consideration to such a move becomes more plausible. Additionally, most students taking data structures are serious about computer science as a major or minor. Conversely, most students taking CS1 and CS2 (especially CS1) do so as a requirement for their major or minor, and the added tools that Ada offers is of little consequence to those who will use mostly menus when interacting with a computer beyond their college days.

Our most realistic assessment is to possibly introduce Ada in our Software Engineering course and our Operating Systems course. The software engineering principles inherent in Ada prod us to use it in the Software Engineering course. However we frequently try to employ real-world projects from our community into this course, and we have yet to find such an Ada based project. (Sioux City, Iowa, has little need for Ada programs or programmers.) The language resources for software engineering are well established for this move, and we shall be easily swayed when such a project from the community comes to our attention.

The more feasible alternative is to use Ada in the Operating Systems course (which is our current plan for 89/90). Given a prior exposure to Ada in data structures, it is quite reasonable to expand on their developed resources and incorporate them into the principles of operating systems. Queues, buses, interrupts, coprocessors, etc. are all easily addressed by Ada. By reusing some of the code written more than just a month or two ago, students will gain a better perspective on the joy in maintenance! As larger projects are tackled, some as team efforts, the concept of working as a software team can also be employed and demonstrated. But it is essential that Ada remain as a tool source, and not be addressed as just another language whose syntax needs to be learned.

Epilogue. Some valuable lessons have been learned from this teaching experience. As easy as it is to teach Pascal in CS1 and CS2 instead of problem solving techniques, it is even easier to attempt to teach Ada as a language rather than as a tool for data structures. But when the focus is maintained on the underlying principles of data structures, the tools that Ada has to offer avail themselves quite naturally to the students. Success is very enjoyable when Ada has been correctly incorporated into the Data Structures course. But warning must be given of the pitfalls that await those who are not fully prepared when entering into this arena.

References

- [1] Harold Youtzy, Jr., "Teaching Ada in a Small College Environment," Proceedings of the Third Annual Eastern Small College Computing Conference, (October 1987), pp. 101-107.
- [2] Grady Booch, **Software Components With Ada**, Benjamin/Cummings Publishing Company, Menlo Park, Ca., (1987).
- [3] Michael B. Feldman, **Data Structures With Ada**, Reston Publishing Company, Reston, Virginia, (1985).
- [4] Norman H. Cohen, **Ada as a Second Language**, McGraw-Hill Book Company, New York, (1986).

TEACHING OLD DOGS NEW TRICKS

Tina Kuhn

317 Calvin Lane
Rockville, Md. 20851

This paper describes the common problems students have learning Ada with regards to their background: C, FORTRAN, Pascal, or COBOL experience. At GTE in Rockville, Md., we have an in-house introduction to Ada course. Currently, over 250 engineers have taken the two week class. All engineers entering the class have some experience in another high-order language. As an instructor of the class, I find the students learn the concepts in Ada not as a function of how smart they are but by the background they have.

The students with a Pascal background, have the easiest time learning Ada because Pascal is one of the base languages of Ada. Yet even these students find many Ada features new and difficult to use: examples include the concepts of packaging, exceptions, private types, and generics to name a few. The students with a real-time background, primarily FORTRAN, C, or Pascal experienced engineers, usually find the concept of multi-tasking easy to understand but hard to use because of the non-deterministic nature of tasks. The students with only a COBOL background have the most difficulty picking up the Ada concepts, mainly because of the extreme differences between the languages and because the engineers usually have not done real-time programming that is helpful in learning tasking. The FORTRAN and C programmers have the most difficulty in the area of strong typing which is foreign in both languages. The strong typing can be a source of extreme frustration for these engineers. Below is a discussion of the Ada specific features and how teachers can tailor an Introduction to Ada course to meet the needs of the students, based on their backgrounds.

PACKAGING

Engineers understand the concept of packaging fairly easily but find that deciding what to put into a package is far more difficult. Indeed, this is a design issue and the subject of many books and papers. Packaging is a critical part of designing a program that is readable, reliable, maintainable, portable, and reusable. Package design should be a part of any introduction to Ada course. Students, in general, are not knowledgeable about software engineering principles. In order to teach packaging and win the students over to Ada, the software engineering principles (abstraction, information hiding, modularity, reusability, portability, verifiability) must be understood and

accepted by the students. The C and COBOL programmers are the most skeptical about the software engineering principles and take the longest time to start thinking in Ada. The COBOL programmers have difficulty with the concept of modularity while students with other backgrounds do not. COBOL programs tend to be large with few units, or subprograms. In addition, there is nothing analogous to a function in COBOL.

STRONG TYPING

Strong typing is the hardest feature to use for students without a Pascal background. Typing errors produce most of the compilation errors and can be very frustrating for FORTRAN and C engineers. Several of the most common problems are outlined below.

1) In the early stages of learning Ada, one of the most common mistakes is a student trying to use a user-defined type as a variable. For example:

```
procedure TEST is
  type LIGHT is (RED, GREEN, YELLOW);
begin
  LIGHT := RED; -- illegal.
end TEST;
```

2) A second problem is a student trying to mix objects that are of different types but that otherwise have the same characteristics. Compounding this problem is the student's failure to grasp the difference between base types and subtypes, and the operations that can be done to each. For example:

```
procedure TEST is
  type BASE_INT is new integer range 1..25;
  subtype SUB_INT is integer range 1..15;
  INT1 : integer range 1..10;
  INT2 : integer range 5..15;
  SUM : integer;
  SUB_INT1 : SUB_INT;
  SUB_INT2 : SUB_INT;
  BASE : BASE_INT;
begin
  SUM := INT1 + INT2;
  SUM := INT1 + BASE; -- illegal
  BASE := INT1 + INT2 -- illegal
  SUM := SUB_INT1 + SUB_INT2;
  BASE := SUB_INT1 + SUB_INT2; -- illegal
end;
```

Another example illustrating the problem of using incompatible types is the following:

```

procedure
  type ARRAY_5 is array (1..5) of character;
  type ARRAY_10 is array (1..10) of character;
  A5 : ARRAY_5;
  A10 : ARRAY_10;
begin
  A10 := A5 & A5; -- illegal
end;

```

SPECIFICATION/BODY

The concept of a specification that can be used and compiled separately from the body of the code takes awhile for students to grasp. The body not being visible to the external sources, and the information hiding that goes along with this, seems to take students awhile to incorporate into their designs. The COBOL programmers have the most trouble understanding separation of body and specification. COBOL programs tend to be large, flat structures with all variables global. Visibility and information hiding issues presented are completely foreign to the COBOL experienced engineers. In addition, the FORTRAN and C programmers, especially those who have frequently used common blocks, find it difficult to understand why the concepts of information hiding and abstraction are good, and hence also have problems with the concept of specification versus body.

VISIBILITY

The whole concept of having to "with" in code in order to get access is different in Ada than in other languages. In FORTRAN, every procedure linked into the executable is visible to every other procedure. In other languages, the compiler does not check the interfaces with other pieces of code. The FORTRAN and C programmers, after learning about the "with" and the "use" clause, tend to "with" and "use" everything available to circumvent good visibility practices. In addition, the concept of an operator not being visible is a side effect of the visibility rules. An example of the nonvisibility of an operator follows:

```

package MY_PKG is
  type MY_TYPE is range 1..10;
end MY_PKG;

with MY_PKG:
procedure TEST is
  Q,P : MY_PKG.MY_TYPE;
  Y   : BOOLEAN;
begin
  Y := P > Q; -- ">" is not recognized.
end;

```


The ">" must be explicitly imported using the "use" clause, fully qualified (i.e. Y := MY_PKG.)"(P,Q)), or renamed. The concept that an operator cannot be used without doing something special is unfamiliar to most programmers.

PRIVATE TYPES

Private types require a special type of design that is foreign to any programmer of another language. The students pick up the syntax of private types but have a harder time seeing why they are good to use, and an even harder time knowing when a type should be made private. To the FORTRAN or C programmer using a private type does not seem worth the trouble. The limitations on the use of a private type often frustrates the engineers.

ACCESS TYPES

The C and Pascal programmers find the concept of access types very easy, but the FORTRAN and COBOL programmers find it very difficult. This is one of several areas where a teacher should learn the background of the students to avoid either boring or confusing the students. Access types needs to be explained to C and Pascal programmers because of the subtle differences between their base language and Ada; but the concept can be taught quickly moving into more complex examples. For the COBOL and FORTRAN programmer, however this is a brand new topic that must be taught slowly and carefully. The concept of linked lists may also be new and confusing. For this lesson, using diagrams and walking slowly through examples seems to help. Some of the problems are:

- 1) Using an incomplete type too early before the complete definition of the access type and node;
- 2) Mixing up the pointer and the node type; and
- 3) Problems with the difference between assigning the access value verses assigning the values inside the type. For example:

```
type POINTER is access A;
   A1,A2 : POINTER := .....;
begin
   A1 := A2;           -- assigning the pointer
   A1.all := A2.all;  -- assigning the values inside the type
end;
```

GENERICIS

Most programmers do not acquire a good understanding of generics until after much experience with the language. The syntax of instantiation is picked up easily but the reason behind the instantiation usually takes awhile to fully understand. The hardest part of generics to grasp is the instantiation of a

generic with a subprogram, especially an operator. This is tough no matter what the student's background.

```
procedure INT_SORT is new SORT (ELEMENT_TYPE => INTEGER,
                                LIST_TYPE    => INT_ARRAY,
                                ")"         => ")"");
```

TASKS

In all other languages, concurrency is done through calls made to system routines provided by the operating system. Tasking is a new concept for all students, but the students with experience in real-time programming find tasking easier. Tasking concurrency and asynchronous concepts are usually picked up easily by FORTRAN, Pascal, and C programmers since most of the engineers have done some kind of asynchronous processing using system routines. Tasking is deceptively complex with many underlying issues. Tasking is nondeterministic and in many cases, the results are machine and compiler dependent. For example, most students assume the selective waits with several open alternatives will be serviced in turn. This is not the case, the LRM states the selection is arbitrary, not fair.

```
loop
  select
    accept .....
  or
    accept .....
  or
    accept .....
  end select;
end loop;
```

In the above example, if all the alternatives are open every time through the loop, one compiler may select the last accept statement each time while another compiler will select the first accept statement. In both cases, the other open alternatives are starved.

The evaluation of guards in a select statement also seems to be a very error prone part of tasking. Guards are evaluated at the top of a select statement each time through the loop. If all the guards are closed and there is no "else" part to the select statement, a program error will occur.

A third problem area is around task termination. Unhandled exceptions in the body of a task can cause a task to silently die leaving other tasks waiting for calls from the hanging terminated task. Tasks in library code that do not have a terminate alternative may hang the program, especially if the task is waiting at a select statement that will never be executed.

EXCEPTIONS

Exceptions and exception propagation are new concepts for engineers of all backgrounds. Most languages use return statuses for the propagation of errors upwards in a program. The concept of exceptions is picked up easily by most students but the exceptions are usually not implemented as efficiently as they could be. Many times exceptions are used as GOTOs. Exceptions and exception propagation problems seem to be evenly distributed among the students. The background of the students does not seem to be relevant. Exception propagation, especially the propagation of exceptions from a library package, seems to cause a lot of programming errors both during compilation and at run time. The students "forget" to put in exception handlers for exceptions propagated from library routines. In addition, students have a hard time deciding how to handle an exception: i.e.; to propagate it upwards again, to capture it and perform some work, or to propagate it upwards and perform some work.

MISCELLANEOUS PROBLEMS

1) Students inevitably try to write to in parameters, or read from out parameters. In COBOL and FORTRAN, parameters are always passed as in out, in C they are always passed as in.

2) Students find the tick mark (') confusing because it is used as both a qualifier and for an attribute:

```
package TEST is
  type FLAG is (RED, WHITE, BLUE);
  type LIGHT is (RED, YELLOW, GREEN);
  procedure MY_PROC (COLOR : in FLAG);
  procedure MY_PROC (COLOR : in LIGHT);
end TEST;

with TEST; use TEST;
procedure MAIN is
  :
  :
begin
  INT := my_array'last;           -- tick mark used with an
                                -- attribute
  MY_PROC( COLOR => FLAG'(RED));  -- tick mark used as a
                                -- qualifier
  MY_PROC( COLOR => LIGHT'(RED));
end MAIN;
```

3) Attributes such as SUCC, PRED, POS, VAL may be applied to subtypes, but the results are identical to applying the attributes to the base types.

```
procedure TEST is
  type DAYS is (MON, TUES, WED, THURS, FRI, SAT, SUN);
```

```
subtype WEEKDAYS is DAYS range MON..FRI;
TODAY : WEEKDAYS := FRI;
MY_DAY : DAYS;
begin
  MY_DAY := WEEKDAYS'succ(TODAY); -- MY_DAY will equal SA
end;
```

SUMMARY

The students with a Pascal background have the easiest time converting to Ada followed, in order of difficulty, by students with backgrounds in C, FORTRAN, and COBOL.

Another factor in teaching the Ada language to experienced engineers is the number of years experience they have in their base language. Typically, the longer an engineer has been working in one language, the more resistance there is to change. The engineers who have spent considerable time using one language are typically "GURUS" in the language and to start all over and lose that "fame" is difficult. Much care and sensitivity needs to be taken on the part of the instructor to ease the engineers transition to this new and frequently overwhelming language.

Process Control Training for a Software Engineering Team

Sue LeGrand, Ann Reedy, Ed Yodis
Planning Research Corporation

Introduction

In the traditional computer science curriculum, students have been taught programming-in-the-small. The standard industry complaint with this approach has been that graduating students lack experience and understanding of what the Software Engineering Institute (SEI) has begun to call the software process [1, 2]. In particular, students are not well acquainted with software life cycle concepts and standards and are inexperienced with a team approach to software development. They lack experience in the application of software engineering concepts and principles within a large project environment. Lately, there has been an effort in the universities to introduce the software engineering disciplines in conjunction with the introduction of Ada. Ada highlights the needs for these disciplines by directly supporting software engineering techniques and by forcing more attention to design issues. The concept of the Ada Programming Support Environment (APSE) also focuses attention on software engineering environments. However, it continues to be difficult to provide large project exposure within the constraints of the university setting. This paper will discuss an approach to offering education and training in the management of software process activities. Some suggestions for the content of courses will be presented as well as an example of an automated software process control system that could be used to assist in the teaching of software process and software engineering concepts.

Students need exposure to the issues and problems of programming-in-the-large (sometimes also referred to as programming-in-the-many). These issues and problems are best seen in the worst case scenario of a large government project with multiple segments; a prime contractor and multiple, geographically dispersed subcontractors per segment; and a system integration contractor or activity. These projects involve hundreds of people; difficult communications across geographic and organizational boundaries and company *cultures*; and large numbers of

change requests. Further, such projects frequently have challenging performance and reliability requirements as well as requirements for an extended operations and maintenance activities that include new technology insertion. Students need an appreciation of the disciplines and effort necessary to collect and collate project, product and process management information within and across projects of this kind. Furthermore, students must experience the difficulties involved with the integration of activities into a disciplined engineering process within a large team or large project. Students should learn how efforts by various teams and subgroups must be coordinated, and how they should be managed throughout the software life cycle.

While students are currently taught about various software life cycle activities in isolation from the rest of the life cycle and about methods, techniques, and generic types of tools to support these activities, they are typically not exposed to the various ways these activities can be combined into a complete life cycle. The students are also not exposed to the disciplines that do not belong to a single life cycle phase or activity, but which stretch across the entire life cycle. These disciplines involve the control and coordination of the life cycle activities, the enforcement of methods, procedures, and standards, and the (logistics) management of the large amounts of product, management, and auxiliary data generated by a large project. Finally, students need to learn about the automation of the software process. This paper will stress a process control approach that has proven useful in real world projects and that may prove useful in an academic environment both as an example of process automation and as a vehicle for providing students better experience with programming-in-the-large issues.

The following paragraphs will discuss the software process issues and the automation issues that need to be addressed in education and training. An example of the type of case study needed to motivate many of these programming-in-the-large issues will also be provided.

Software Process Issues

Students need exposure to the entire software life cycle from a process point of view. They need to see the software process as a team effort and as an engineering effort requiring the applications of discipline and sound planning and management techniques. This view is necessary to counteract the impression of software development as a handicraft that is given by teaching programming-in-the-small. Many computer science or software engineering students enter courses

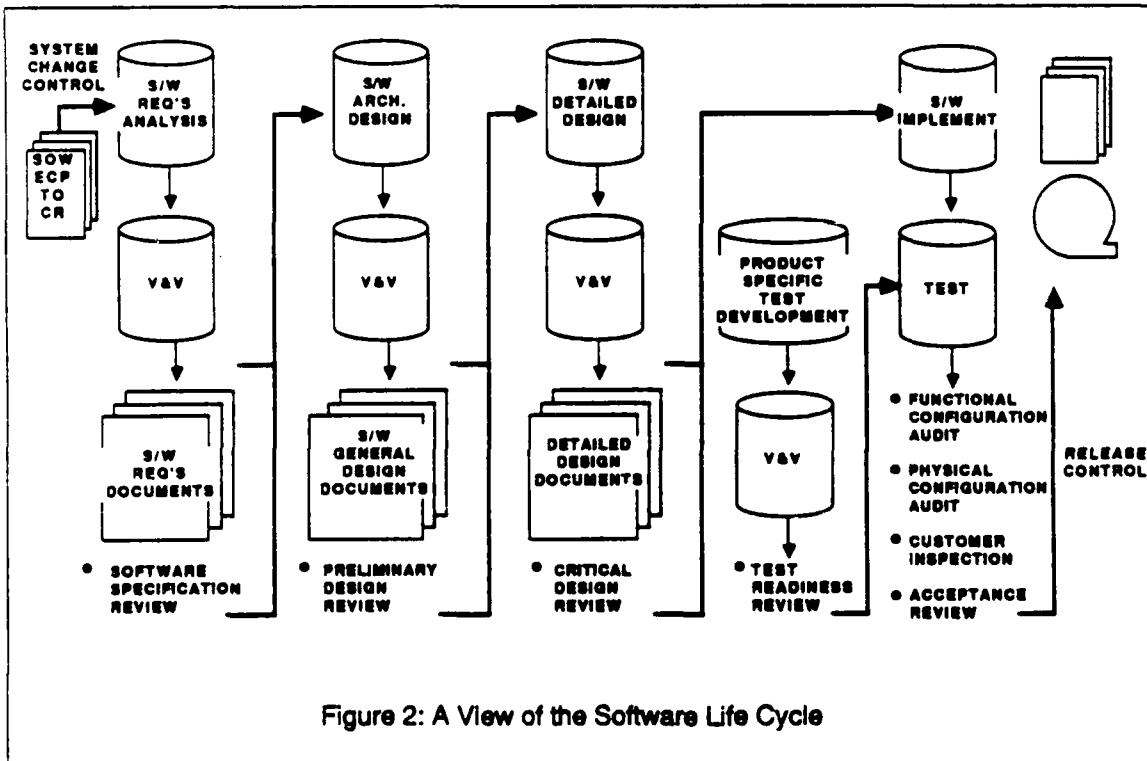
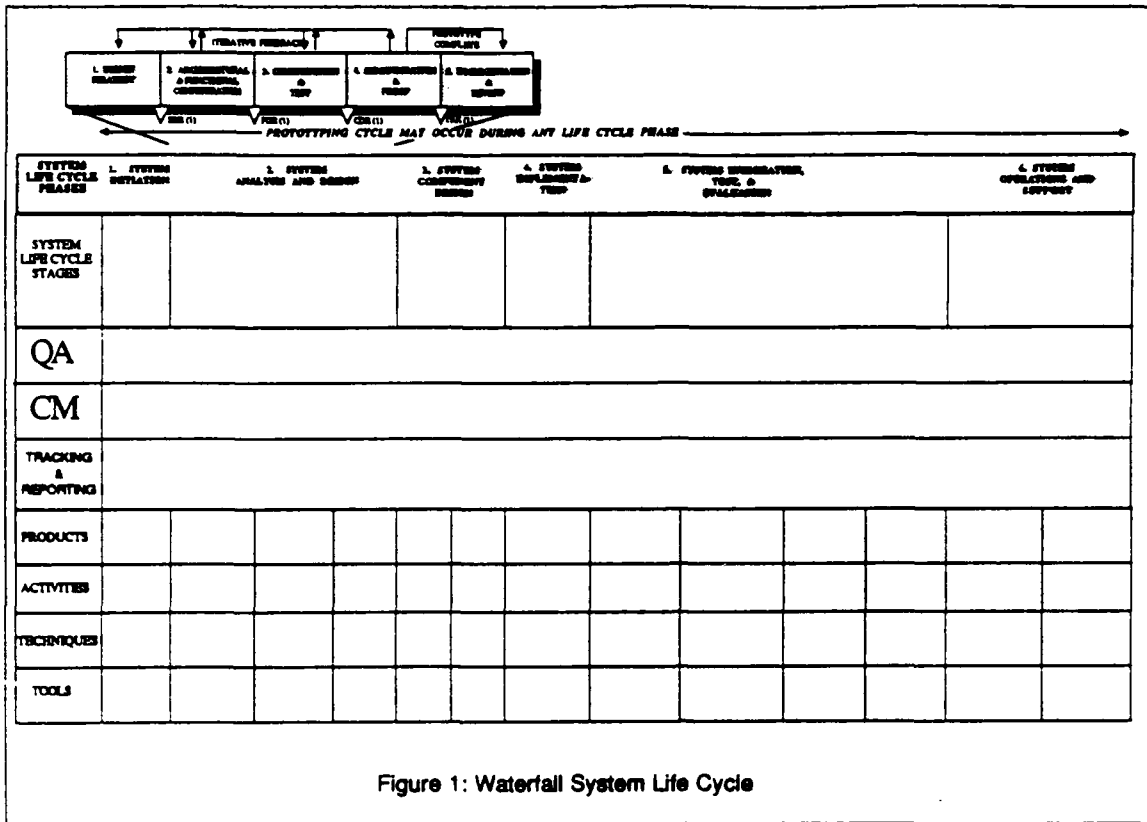
thinking that the software process consists solely of the implementation phase of the life cycle (i.e., "programming"). They need to be exposed both early and often to the software process concepts discussed below.

Students need to learn the concept of the software life cycle. The software process includes many interrelated activities. There are the basic phases of the life cycle, including requirements analysis, design, implementation, testing, operations, and maintenance. Students need to be exposed to the various ways these phase activities can be combined together to form a life cycle model. Examples include the classic waterfall model, the waterfall with prototyping, and the spiral model. Figure 1 shows a classic waterfall model skeleton where the software life cycle is contained within the system life cycle. Figure 2 shows a different view of the software life cycle that emphasizes test and evaluation, auditing, and verification and validation (V&V) activities within each phase and views maintenance and changes as additional passes through the original life cycle. Students need to be aware that there are various standards for life cycles.

When viewing the software life cycle from a process point of view, the students must be focused on those activities and functions that stretch across the entire life cycle or that result from the combination of the phase oriented activities into an integrated whole. These activities or functions include software configuration management (SCM); quality assurance and engineering (QA); status tracking and reporting; management planning, control, and coordination; and logistics support.

Students need to be introduced to SCM, both in terms of its subdisciplines (configuration identification, change control, configuration status accounting, and configuration auditing) and in terms of the separation between policy (i.e., the number and structure of control boards, etc.) and enforcement. The view of SCM should not be restricted to studying the formal baselines required by standard life cycles but should include the integration of SCM into the software process.

In their introduction to QA, students, again, must not be restricted to studying the formal reviews required by some standard life cycle but must include the verification and validation activities required within each phase of the life cycle to insure that quality is "built-in." (Figure 2 illustrates these activities.) Students must learn to separate QA policy issues (requirements for test plans, regression testing, etc.) from enforcement or test management issues (i.e., ensuring that all



required tests are performed, regression testing is performed, etc.). Students must be exposed to the interrelationships among the various life cycle wide activities. For example, the QA policy with respect to testing will impact the SCM function since test baselines will need to be managed for each test procedure.

Among the life cycle wide activities, the management activities of tracking and reporting and the functions of planning and logistics are among the most important and among the least understood in the industry today. Figure 3 illustrates the range of topics that must be covered by management planning for the software process. It is a management responsibility to perform the planning for the entire software process and to lay out in detail how the process is to be carried out. This task includes not only the classic management functions of personnel planning, task definition, and scheduling but also the function of setting policy (i.e., for SCM,QA, etc.), standards and procedures (i.e., enforcement mechanisms), as well as methods and tools. It is a management responsibility, for example, to ensure that there are smooth transitions between the methods chosen for the life cycle phases.

Management is also responsible for insuring that the software process that has been planned can be effectively executed. The logistics support required for proper SCM, QA, and software development method implementation is high, and this cost usually means that automation must be used. If this is the case, then management is also responsible for insuring that the automation tools and techniques selected properly support the desired activities or functions and that these tools are properly integrated. Students must be made aware of these management responsibilities, especially since failure in these management areas is currently a frequent cause of problems in industry.

Case Study Example

The Space Station Freedom Program provides a good example of a case study. This NASA program has all the characteristics mentioned above for a large government project in which the software is a critical component. In this case, there are four prime Work Package contractors, each with multiple subcontractors. Each prime contractor is being managed by a different NASA Center, and the resultant software products are being integrated at Johnson Space Center. The Space Station Freedom Program is unusual in that it contains the full range of types of software applications, ranging from embedded, real-

MANAGEMENT AND RESOURCE CONTROL	BUDGETARY CONTROL																							
	SECURITY MANAGEMENT																							
	TASKING, SCHEDULING, STATUS REPORTING																							
	CONFIGURATION MANAGEMENT																							
QUALITY CONTROL/REVIEW	REVIEWS AND AUDITS																							
	TRACEABILITY																							
T&I CONTROL	UNIT, COMPONENT, SUBSYSTEM, SYSTEM LEVELS																							
TECH. CONTROL	DEVELOPMENT METHODS, STANDARDS, TOOLS USE/ENFORCEMENT																							
PHASED TECHNICAL ACTIVITIES	LIFE CYCLE OF SOFTWARE																							
	<table border="1"> <tr> <td>PFE</td> <td>SOFTWARE</td> <td>PRELIMINARY</td> <td>DETAILED</td> <td>CODING</td> <td>SYSTEM</td> <td>DEPLOY.</td> <td>POST-</td> </tr> <tr> <td>SOFTWARE</td> <td>REQTS</td> <td>DESIGN</td> <td>DESIGN</td> <td>AND UNIT</td> <td>INTEG. &</td> <td>TESTING</td> <td>DEPLOYMENT</td> </tr> <tr> <td>DEVELOPMT</td> <td>ANALYSIS</td> <td></td> <td></td> <td>TESTING</td> <td>TESTING</td> <td></td> <td>SUPPORT</td> </tr> </table>	PFE	SOFTWARE	PRELIMINARY	DETAILED	CODING	SYSTEM	DEPLOY.	POST-	SOFTWARE	REQTS	DESIGN	DESIGN	AND UNIT	INTEG. &	TESTING	DEPLOYMENT	DEVELOPMT	ANALYSIS			TESTING	TESTING	
PFE	SOFTWARE	PRELIMINARY	DETAILED	CODING	SYSTEM	DEPLOY.	POST-																	
SOFTWARE	REQTS	DESIGN	DESIGN	AND UNIT	INTEG. &	TESTING	DEPLOYMENT																	
DEVELOPMT	ANALYSIS			TESTING	TESTING		SUPPORT																	
LOGISTICS SUPP'T	TRAINING, OPERATIONS, MAINTENANCE PLANNING/EXECUTION																							

Figure 3: Software Process Planning Requirements

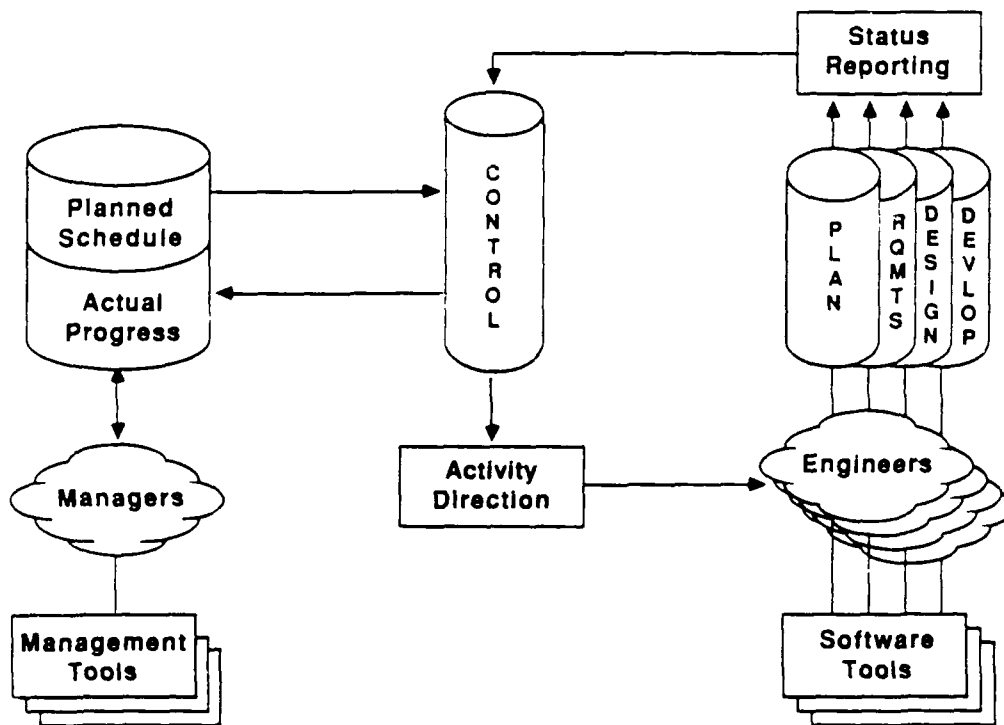


Figure 4: Process Control

time, life and property critical systems to information systems. The Space Station Freedom is an example of a "never ending system" since it has a very long planned life cycle during which it will continue to be modified and expanded. Since future requirements cannot be fully anticipated, few assumptions can be built in to the system design regarding limits on memory and processing power. The system will be multi-vendor, and, as technology evolves, it must be possible to replace both original hardware and software with items from different vendors or contractors. The system will be maintained by yet other contractors or vendors.

The study of the requirements for programs like the Space Station Freedom Program can give students the background necessary to understand why software development and maintenance must be treated as a disciplined engineering process. In the context of case studies, the need for layered software architectures, standards, and carefully designed interfaces becomes clear.

Automation Issues

The software engineering student needs experience both in working within a disciplined environment and in working in a team and project oriented environment. The disciplined environment should reinforce the rigorous and systematic approach to software development that the student is trying to learn. This environment should also reinforce professional attitudes and behavior. The environment should provide rapid feedback to the student. This feedback can sometimes be supplied by tools that support a specific method but feedback in the academic setting is usually supplied by the instructor. In project oriented classes, this feedback is frequently restricted to the final grade on the student's project, since logistics issues make it too difficult for the instructor to perform many in depth intermediate checks on the student's progress without seriously impacting the student's work. In courses where a small team approach is used, the instructor has difficulty in distinguishing the contributions of individual students and the students spend much of their time in trying to coordinate their activities.

Students need to learn the concepts of software engineering environments and need exposure to at least one working software engineering environment. These environments are becoming more and more common in the software industry. Automated software engineering environments are becoming a requirement in many government requests for proposals. A software engineering

environment based on automated process control is in place to support the Space Station Freedom Program (SSFPO) [5], and similar environments are being investigated by such organizations as the Air Force Space Command and the Space Defense Initiative Office (SDIO) [3].

The needs of students are also shared by many entry level employees in industry who are undergoing on-the-job-training in company or project specific methods and techniques and who require the same types of fast feedback and positive reinforcement as academic students. In the hardware engineering world, an apprentice style relationship is frequently set up between a senior engineer and a new junior engineer. The designs prepared by the junior engineer must receive the review and signature of the senior engineer, who reviews them for adherence to standard practice and project requirements, before the plans are forwarded for configuration management and additional quality review. Unfortunately, this approach does not seem to be used within the software industry. The nearest equivalent would be peer reviews or walkthroughs, but in too many cases, even this type of detailed review is lacking.

The approach to a software engineering environment described in the next few paragraphs is based on a process control approach to the software process. The approach described has been used successfully on working projects and has been successful in helping new employees become productive rapidly in project despite their lack of familiarity with the programming languages, methods, standards, and the environment itself. This process control approach also meets many of the needs of academia.

Process Control

The process control approach to a software engineering environment that has been implemented at Planning Research Corporation (PRC) is based on a separation of concerns between the functionality of the environment framework and the functionality of software development tools. The framework provides the control and coordination for activities; the enforcement of standards and policies; and a repository for support information as well as the software products. The framework automates the labor intensive overhead and administrative functions of the process. The tools assist the project members in the actual creation or modification of the products (software and associated documentation, i.e., software in all its representations).

This approach can best be understood by analogy with a manufacturing shop floor control system. The control system manages and controls the manufacturing process as the product components (in this case software modules and documentation) move about the shop floor from toolstand to toolstand. In manufacturing, retooling is often necessary if the organization wishes to remain competitive. This retooling may take the form of replacing the tools available at one toolstand with other tools, or it may take the form of a major reconfiguration of the shop floor and the process flow. In software development, this retooling is also necessary to take advantage of improvements in automated tools, methodologies, techniques, and computer hardware. The control system can be configured for the initial shop floor configuration and later reconfigured as part of the retooling process.

Figure 4 illustrates the approach. Managers develop project plans using available management tools and load the life cycle phases, products, schedules and low level tasks into the system (i.e., they configure the control system). A potential project configuration is illustrated in Figure 2. The developers (i.e., analysts, designers, implementors) develop product parts in response to their tasking, using the available tools. Testers (i.e., quality assurance, verification and validation, configuration management, or formal integration and test personnel) perform V&V and testing tasks in response to their tasking. The control system captures their products and status as part of the natural flow of their work (i.e., as they check in and check out product parts, build test beds, post test results, etc.). The actual progress of the project is available in real-time for review by the managers and comparison against the project plans. When discrepancies occur or change requests are approved, then the managers can adjust the plans.

Benefits

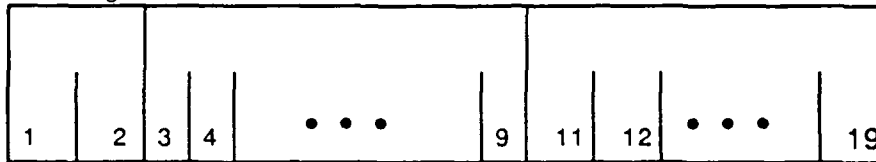
This approach enforces a disciplined project or team view of software development. Planning must be done before the framework can be configured. Since the framework manages the product configuration and holds the *official* copy, there is less of a tendency for a programmer to view a program as *his*. Testing (or V&V) is required for all products, and the framework enforces the configured test plans and procedures.

Figure 5 shows some example tests or verification and validation procedures that have been applied in a PRC project. In the PRC implementation, there are three required types of testing: component level testing (for static testing and standards enforcement); integration level testing (for functionality in code or content in

**Preliminary Design
Component Level Tests**

- Correctness of Notation
- Adherence to Method, Procedures, and Standards
- Appropriate Degree of Completeness
- Consistency with Referenced Design Components (Compilation)
- Traceability to and Consistency with Requirements
- Performance Requirements Documented or Suballocated
- Correct Use of Specified Tools

Component Testing Integration Testing System Integration Testing



- Adherence to Detailed Design Specifications
- Traceability to Detailed Design Specifications and Called/Calling/Imported Modules
- Adherence to Coding Standards, Implementation Methods, and Procedures
- Consistency (Compilable)
- Completeness (Executable with Appropriate Stubs and Drivers)
- Correctness of Functionality
- Performance Acceptable with respect to Performance Level

**Implementation
Component Level Tests**

Figure 5: Example Tests for Methods and Standards

documents); and system level testing (for performance and stress testing in code and final review in documents). This testing is performed by a specialized organization in large projects and is distinct from the unit testing that the individual developer performs. The control framework ensures that all product parts undergo the required testing. Feedback from the testing process is rapid. The framework coordinates the activities, removing the need for direct person to person coordination meetings. The newly developed or corrected product part is made available for formal testing as soon as it is checked in. The tester can track its progress and readiness through real-time, on-line reports. Thus, ego-less programming or writing is encouraged. Since all team members can review the project progress reports on line, there is more incentive to work toward project goals.

Figure 6 illustrates the types of management status and reporting information available on-line from the process control framework. This figure also illustrates how the process control framework supports the life cycle wide disciplines of configuration management and quality assurance. The framework allows all types of managers to track the progress of the project without interrupting the workers. In an academic environment, this same approach can be used to allow the instructor to track progress in a classroom project and to pinpoint those students who are creating bottlenecks (i.e., who are impacting the work of others by being behind schedule on their particular tasks) without using classroom time on status meetings. This type of process control framework would also allow students to act as managers of different types (i.e., project managers, configuration managers, or quality managers), while the instructor took the role of client or end user of the developed system.

This framework supports the entire software process since the life cycle paradigm used views changes and maintenance as additional passes through the configured phased activities. The framework stores the products, their traceability, and their history, so that the information needed to perform impact assessments and begin planning for changes or maintenance fixes. This approach could be an advantage for academia as it is for industry. If the framework is used to control a product set, then one class can easily pick up where the last class left off and continue enhancement of a software product set. This type of approach is useful in giving software engineering student experience in the type of maintenance projects that are usually encountered in

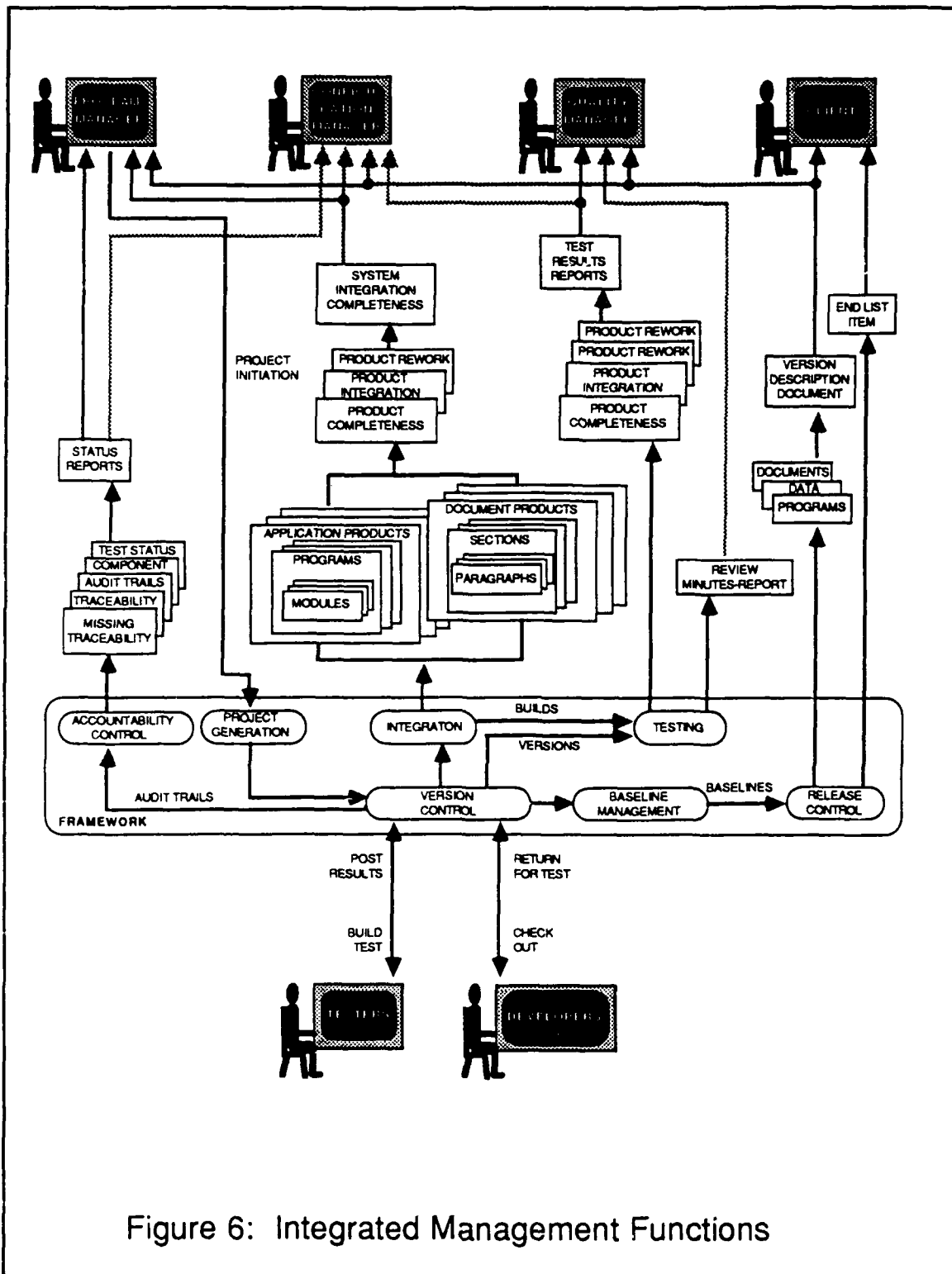


Figure 6: Integrated Management Functions

industry. More details on the functionality of the PRC implementation will appear in [4].

Architectural Issues

In order to be useful in either academia or in industry, a process control framework must be implemented using the same architectural principles that it will be used to enforce. That is, it must be based on a layered architecture that allows full exploitation of existing hardware and software resources. The framework architecture must permit distribution of functions, accommodate heterogeneous hardware configurations, and exploit available communication facilities. The control framework must be easily transportable to new hardware and operating systems at reasonable cost. The environment database, including the life cycle products and their relationships and attributes, must be easily moved between framework implementations. There must be no performance penalties for using the framework. It must cooperate at some level with existing systems to take advantage of their security and performance features. Finally, the framework must allow the use of existing software tools and allow flexibility for retooling as necessary. The framework architecture can itself be used as a case study. At PRC, the framework prototype was used to develop the production version, and the framework is used to control and maintain itself.

Summary and Recommendations

There is an urgent need to teach software engineering students, computer science students, and software management students about the software process and programming-in-the-large. This need has been highlighted by the advent of Ada. The use of the language has forced projects to spend more time and effort in the up-front design and planning stages of the project. The use of Ada also punishes sloppy or poor configuration management with large recompilation penalties. The advent of Ada has also fostered the automation of the software process and the introduction of the concept of a software engineering environment.

The need for a structured software process and software engineering disciplines should be motivated by the use of case studies at all level of software engineering education. The concept of the software process should be introduced in a required introductory overview class. The use of a process control framework based environment to control all classroom project activities will reinforce the discipline that the

student is currently studying while giving the students the experience of working in a orderly professional manner. There should be an upper division/master's level graduate course devoted to detailed study of the software process and the concepts of software engineering environments. This course should address the role of life cycle wide disciplines such as configuration and quality management within the software process and the concepts of process control.

The software process must be taught with other software engineering concepts in university curricula in order to provide complete life cycle software engineering education to support the never ending systems required by NASA and other organizations building large, complex, distributed systems.

References

1. Humphrey, W.S. "Characterizing the Software Process: A maturity Framework. *IEEE Software*, March 1988, 73-79.
2. LeGrand, S., G. Freedman, L. Svabek. *A Report on NASA Software Engineering and Ada Training Requirements*. Research Institute for Computing and Information Systems, University of Houston at Clear Lake: November 15, 1987.
3. LeGrand, S., A. Reedy, F.C. Blumberg. "Process Control of a Complex Computerized System". Presented to the Joint Applications in Instrumentation Process and Computer Control Symposium: March 23, 1989.
4. Reedy, A., F.C. Blumberg, D. Stephenson, & E. Dudar. "Software Configuration in the Maintenance of Ada Software Systems". To appear in *Proceedings of the Conference on Software Maintenance-1989*. IEEE-CS Press.
5. *System Concept Document: Space Station Software Support Environment Revision 2.1*. NASA Johnson Space Center: July 7, 1988.

ADA TRAINING AT KEESLER AIR FORCE BASE

*Captain Roger D. Beaman
3390th Technical Training Group
Software Engineering Branch
Keesler Air Force Base, MS*

Abstract:

The government, academia and industry have established many education and training programs to teach Ada and software engineering. The Software Engineering Training Branch at Keesler Air Force Base has been providing Ada training throughout the federal government since 1984. This paper outlines the progression of that training through the years, from early methods to our current training program. It also includes a discussion on how we conduct our courses, how we train our students, how we assure quality of our courses and instructors, and who we teach. In addition, this paper brings to light some of the unique problems our instructors encounter in a mobile training environment.

Background:

With the mandating of the Ada Programming Language for government systems in the early 1980s, the Air Force Air Training Command (ATC) initiated actions to obtain funding for manpower and equipment to develop an Ada training program for the Air Force. Funds were obtained from a large Department of Defense (DOD) command and control project called Worldwide Military Command and Control Systems (WWMCCS) Information System (WIS) for the initial equipment to support what was then known (late 1983) as the Ada Training Section. As the need for Ada training increased among the services, ATC took the lead in providing that training. They fully funded our Ada training effort and supported our course enhancements to include a strong emphasis on software engineering within each course. Today, our Software Engineering Training Branch provides a product recognized throughout the DOD as some of the best Ada software engineering training within the federal government. With more requests for our courses than we can handle, we are working to expand the availability of this training throughout the government by working hand-in-hand with several agencies; training their trainers, and helping them establish their own Ada Software Engineering training programs.

What We Teach At Keesler Air Force Base:

Our early mission was to indoctrinate our students on the need for Ada and the fact that Ada was not just another programming language [1]. We believed our job was to provide different levels of education/training depending on the level of student. We focused on; (1) a familiarization of the language and concepts targeted for executives; (2) an overall introduction to the language, including coding constructs, for high-level software managers and support personnel; (3) an introduction to the fundamentals of the language for program managers and supervising software engineers; and (4) detailed language training targeted to the needs of programmers. We strongly believed all personnel who dealt with software should be educated for the DOD Ada transition effort to succeed.

Our familiarization training, "Ada Executive Orientation", concentrated on the need for technology to solve what was coined "the DOD Software Crisis" [2]. We discussed software engineering, the history of Ada's development, and the need for an Ada Programming Support Environment (APSE). In addition, we amassed a list of ongoing activities within the Ada community to show the ever-growing emphasis on the language within the DOD. The purpose for providing an orientation to executives was to generate top-down support for the Ada transition.

We recognized most decisions on software projects rested with software managers and their support staff. An "Ada Managers Orientation" course was developed which expanded our executive course to include specific details of language constructs along with their support for the goals and principles of software engineering [2].

Our "Ada Project Manager" course was an 80-hour training session allowing managers to experience Ada programming and Object Oriented Design as described by Grady Booch [2]. It presented a high-level overview of the major language constructs, and dedicated over 40 hours to hands-on lab time. This allowed project managers a chance to sample the language.

Finally, our "Ada Applications Programmer" Course (6-weeks in length) was devoted to training programmers in all aspects of the language and details within the Language Reference Manual (LRM). It was also designed with over 50% hands-on lab time, allowing the programmer to explore the language and develop an appreciation for it's power.

Our early view (1984) of presenting Ada was very successful and well received, but our emphasis was on the language as a means, instead of as a tool for productivity. It was hard to present anything other than a syntax course.

Through a continual review process, our emphasis and our training philosophy changed drastically from our initial courses in 1984. Then, we saw our job as selling the Ada Programming Language. Our discussion of software engineering was superficial at best. In addition, our programmer course was too long (6 weeks); it's length dictated by the slow hardware and subset compiler we originally used. However, we continued to revise our courses to match the needs of our students and we obtained funding to upgrade our hardware and purchase validated compilers. By 1986 our courses were extensively revised. We concentrated on the need for software engineering and the support Ada's constructs provided for implementing system designs, instead of emphasizing the history behind the development of the language. Our orientation courses stressed the management issues of an Ada transition and, thanks to the new hardware and compilers, we reduced our programmer course to four weeks. Both application courses evolved into well-designed, intense offerings and were accredited through the Community College of the Air Force (CCAF) at 3 and 5 semester hours respectively. We began to see a dramatic increase in requests for our training.

Today, we offer only one orientation course, a 2-day course, "Orientation to Ada Software Engineering", designed to address management concerns on the risks and benefits of transitioning to Ada. Both of our application courses now devote almost 60% of course time to hands-on coding projects. In addition, we stress software engineering throughout as the means to increase productivity. (A course description and outline of each is in Appendix A.) As our training matured, we received more and more requests for our courses. During fiscal year 1988 (October 1987 through September 1988) we taught 768 students. This year (fiscal year 1989) we expect to train 1400 students.

How We Conduct Our Courses:

We conduct our courses in one of two forms; either resident or through the Mobile Training Team (MTT) concept. Resident courses are much the same as courses taught on a college campus. Students from throughout the country (and sometimes throughout the world) come to Keesler Air Force Base, Biloxi, Mississippi to attend class at our Computer Training Center. The students either reside in Keesler's temporary lodging facilities or at near-by motels. In our training center,

we have both formal classrooms and complete labs to support our courses. We differ from a campus approach, however, in our class format.

Unlike college students, our students are full-time employees of the federal government. The longer we keep them in class, the longer they are away from their primary job. Because of this, we teach on an 8-hour day, 5-days-per-week schedule. Our class size is limited to 12 students. Teaching in this environment puts great pressure on the instructor to constantly assess the level of achievement each student attains. Typically, because of the volume of information we present in our courses, our instructors spend 8-12 hours overtime per week working with those students who have problems assimilating the material.

While bringing students to Keesler AFB to attend our courses in residence removes them from their daily distractions and increases concentration in the class, it's more cost effective for the government to send one instructor to teach 12 students on-location. Courses taught by Keesler instructors at locations requesting training are considered MTT deployments. Because of the obvious cost savings of an MTT class, most of our instructors travel from 50-65 percent of the year. It becomes a full time job maintaining the deployment schedule and keeping in contact with deployed instructors to assure each class receives the same quality training as it would on a resident basis. However, MTT deployments also create unique problems for the instructor.

When you teach courses in your own classroom and lab, you become intimately familiar with your lab computer, editor, and compiler. Most individuals in this field know how annoying it is to transition between one computer system to another, not to mention learning to use a new editor. When you're a mobile training instructor however, you learn first hand that a compiler validation certificate does not ensure quality; compilers are not born equal. From cryptic error messages to implementation of language features that just don't work, compilers at each class location provide unique challenges. In addition, our instructors may not have experience with the compiler and editor available to support the class (there are well over 150 validated compilers on the market). In many cases, they spend the first few days of each course learning the system and compiler along with the students.

Problems also exist with the hardware reserved for our classes. With the advent of several Ada compilers for PCs, many offices who purchased PCs to host an Ada compiler failed to realize a math co-processor may be required to support operations for floating point calculations. MTT instructors must often adapt lab exercises to use other than Float, a job not always as easy as it appears. Whatever it takes, it's the instructor's job to satisfy the objectives of each course.

When we send an instructor to teach an MTT course, we require the host location to provide administrative support, a classroom, and a separate terminal per student to support the course. However, in real life, the location of classroom and lab may be separated by miles, computer resources may only be available during evening hours, lighting conditions in the classroom may not be conducive to the use of 35mm slides (our course material is in 35mm slide format). Also, administrative support may not be up-to-par causing our mobile instructors to become teacher and clerk. Other influences such as illness combine to make mobile training a unique experience.

Of all the headaches associated with software and hardware, none is more difficult to cope with than teaching students who don't possess the prerequisites for the course.

How We Train Our Students:

Our courses are not basic programming courses. To accomplish all we've programmed for each class, we require our students to be experienced programmers. Since we can't enforce these

prerequisites, many times we encounter the exact opposite. These students have difficulty coping with our courses because of the pace of the class.

How we teach is best described as a fire-hose approach, introducing a consistent flow of new material throughout a compressed training schedule. This places a great responsibility on the student to learn the material. Because of the amount of information we convey in the short amount of time, we do everything possible to assist our students in assimilating the material. (Ultimately, we need to reap the most return per training dollar spent.)

Our courses are slide-intensive. We rely heavily on examples to reinforce the basic concepts taught. To reduce the time necessary for note-taking, we provide our students a copy of all our slides in a student handout. We reinforce the concepts we teach with hands-on application projects. This becomes the primary evaluation tool for our instructors to assess each student's progress and understanding.

Lacking experience in PASCAL or other high-order languages doesn't mean a student can't pass our courses. Of all the classes we've taught, our drop-out rate is less than 10% and our failure rate is less than 1%. However, some classes are more demanding than others.

A challenging assignment is teaching a class of assembler programmers. A class of assembler programmers almost assuredly means many hours of overtime and student counseling because our instructors end up teaching basic data structures along with the advanced Ada concepts our courses are designed to teach. We also have difficulty teaching some experienced COBOL programmers. Understanding the concept of strong typing gives many COBOL programmers headaches. But the most difficult job is training students with little or no programming background.

You'll notice in Appendix A, both our 9-day and 4-week courses require students to have previously learned programming skills, ranging from a fundamental knowledge of programming concepts, to definitive requirements for a working knowledge and recent experience in a high-order language. Yet our instructors teach many students that don't meet these requirements. As in any industry, organizations with inexperienced staff are sometimes tasked to develop software systems. For these classes, our instructors literally go out of their way to assure success. Our low drop-out and failure rates attest to their ability.

Since over 80 percent of our students are trained through the MTT concept, how we develop and control our courses, and how we train our instructors is extremely important. Our goal is to assure the same quality of training is achieved regardless of the training site.

Our Assurance of Quality Training and Instruction:

All of our courses are developed using the United States Air Force Instructional Systems Development (ISD) Process [3]. This is a 5-step process:

1. Analyze system requirements
2. Define education/training requirements
3. Develop objectives and tests
4. Plan, develop and validate instruction
5. Conduct and evaluate instruction

For course control and continuity of instruction, we place heavy emphasis on steps 4 and 5. Since we operate primarily in a mobile environment, validating our courses and evaluating our instructors is the only way we can control the quality of our Ada Software Engineering training. We stress

development of lesson plans and rigorously control all student handouts and visual aids. In addition, when we put a new version of our course on-line, we follow a thorough review and critique process to evaluate the effectiveness of the course. Students are also required to critique each class which further helps us assess our training and identify areas of needed improvement. It's this ISD process which has helped us mold our courses into the products we provide today.

Our method of course development not only assures the quality of our courses, but also assists us in assuring the quality of our instructors. The best course in any subject is only as good as the instructor presenting the material. We're extremely proud of our instructors and believe their quality is a direct result of our own rigorous instructor training program and course critique process.

All new instructors assigned to Keesler AFB take a 5-week Technical Training Instructor Course (TTIC) designed to teach a person the skills necessary to teach. During this course, instructors learn how to construct lesson plans, how to counsel students, and in many cases, how to speak in front of a classroom. They must complete this course before being assigned a teaching job.

Typically, most new instructors are also new to the Air Force. Our needs for experienced data professionals are offset by the competing needs of other Air Force organizations. Consequently, we become the training ground for young officers with little-to-no actual on-the-job experience. Fortunately, most new instructors have studied several languages in college and can achieve a smooth transition to Ada. Unfortunately, relating to software engineering is more difficult.

As we've revised our courses, we've also revised our instructor training program to capitalize on efficiency. During our first attempt at Ada training, we concentrated on teaching each new instructor the management implications of developing Ada systems. This approach required an extensive reading and individual study program, however it developed an instructor with the broad background necessary to field questions from senior management on Ada transition issues. It also took 8-12 months following TTIC to produce an instructor that could teach both orientation and application courses. This wasn't a problem in the beginning because our efforts were concentrated on educating management on the benefits of the language. But as requests for application training exceeded the need for orientation courses, the requirement to expedite our instructor training program magnified.

Today, because of our high course demand, we need our instructors productive as soon as possible. We've slashed that 8-12 month training period by concentrating on developing language skills first. Each new instructor attends both of our application courses as a student trainee. We then assign an experienced instructor to work with the trainee to answer any questions and to help assess that trainee's potential. After going through our courses as a student, trainees are assigned supplemental coding projects designed to fully acquaint each with the nuances of the language. In addition, we try to schedule new instructors for software engineering seminars and tutorials to further enhance their background of program development issues. When the trainer is satisfied the new instructor has a grasp on the language, he directs the instructor's effort in building a personalized lesson plan. It's at this stage we begin to assure continuity for our courses.

As the trainee develops the lesson plan, the trainer conveys the purpose of each lesson, reviews all visual aids with the trainee, and discusses the purpose for each 35mm slide. Each new instructor must understand the essence of each lesson in order to preserve course continuity in a mobile environment. But to ensure that understanding, all new instructors teach their first MTT course with an experienced instructor monitoring the class. Each evening, the trainer discusses the day's lesson and provides the trainee with constructive feedback, developing the confidence needed to teach that first mobile course - alone. Even after the instructor is qualified to teach, periodically we

formally evaluate our instructors and course critiques to assure overall quality is maintained. This is the final link toward ensuring course continuity.

Through this approach, we've refined our instructor training program so that once an instructor is assigned to us after TTIC, he's ready to teach in 3-4 months. This means we add a new instructor to our staff in less than half the time it previously took, and we reap a 20% increase in instructor productivity over a normal 4-year assignment. In addition, we've found after the instructor teaches several application courses, learning to teach our orientation course becomes an easy transition.

Who Are Our Students:

We're extremely proud of our staff and courses. Over the past several years, other services and government agencies have used our courses as the basis for establishing their own Ada training. The Air Force Strategic Air Command (SAC) used our Ada Application's Programmer course as the basis for establishing their own in-house training program. The United States Marines requested more courses than we could provide. Instead, we trained their staff so they could establish their own Ada training curriculum at their Computer Science School in Quantico, Virginia. We continue to support the United States Army training effort and have scheduled three additional courses this summer. We are in the process of providing training to the Federal Aviation Administration (FAA) and also provide the United States Navy with needed support. Additionally, we've provided two courses to the German Air Force to assist them in establishing their own Ada training and have received contracts to provide training to the North Atlantic Treaty Organization (NATO) in Brussels, Belgium as soon as they procure the needed computer support.

Throughout our short history, we've accomplished our training mission with a staff of 15 dedicated instructors. We've consistently applied what we've learned about how to train, how to cope with equipment and software problems, and how to maximize student learning toward increasing the efficiency and effectiveness of our small staff. We expect to continue revising our courses to meet the needs of the Air Force; but with only 15 instructors authorized, and with the ever increasing requirements for Ada training within the federal government, our efforts seem best spent in a "train the trainer" role.

Appendix A:

NAME: ORIENTATION TO Ada SOFTWARE ENGINEERING

DURATION: 2 Days

LOCATION: User's site using Mobile Training Team (MTT)

OBJECTIVE: Training familiarizes managers with issues affecting the transition to Ada. The role of software engineering, to include the goals and principles of software engineering, is looked at with respect to the life-cycle implications of software in the DOD today. The language features that set Ada apart as a programming language are addressed with their relation to software engineering. Functional design methodologies are compared to Object Oriented Design for use with Ada systems. Discussion of DOD policies and standards will familiarize managers with issues surrounding the current integration of Ada within the DOD. Other issues covered include the need for management commitment, training issues, compiler issues, software reuse, hardware concerns, and configuration management issues. The manager will understand and appreciate the advantages, as well as risks, involved with the transition to Ada within the DOD.

PREREQUISITES: None

RECOMMENDED FOR:

Senior Executives
Mid to Upper Level Managers
Project Managers

SYLLABUS: Impact of the Software Crisis on the DOD
Relationship of Software Engineering and Ada
Design Methodologies
DOD Standard Considerations
Managing the Transition to Ada
Management/Training Issues
Software/Hardware Issues
Highlight Risks and Benefits

NAME: FUNDAMENTALS OF Ada PROGRAMMING/SOFTWARE ENGINEERING

DURATION: 9 Days

LOCATION: Keesler Technical Training Center or MTT at Users Site

OBJECTIVE: Training familiarizes programmers and managers with the fundamentals and benefits of the Ada programming language. Heavy emphasis is placed throughout this course on how Ada supports the goals and principles of software engineering and how students can apply sound software engineering techniques. Over 50% of the course is dedicated to hands-on programming using the fundamentals of each of the Ada language concepts. The use of an appropriate design for Ada is stressed throughout the course and students are given exercises to practice these design techniques. Software engineering, design, and coding principles are brought together and applied in a comprehensive final project. Graduates will be able to design and code simple Ada systems on their own and evaluate Ada code written by others.

PREREQUISITES: Students must have a fundamental knowledge of programming concepts. MTT requires a host site to have a validated Ada compiler with one terminal per student. Class size is limited to 8-12 students.

RECOMMENDED FOR: Project Managers
System Configuration Managers
Design Consultants
Programmers

CCAF Credit: 3 Hrs

SYLLABUS: Fundamentals of Ada Systems
Software Engineering
Language Features
Program Library
Simple Control Structures
Simple Input/Output
Basic Ada Types
Purpose of Typing
Type Declarations
Classes of Ada Types
Control Structures
Structured Programming
Sequential/Conditional
Subprograms
Procedures
Functions
Packages
Specifications
Body
Private Types
Applications
Exceptions
Generics

Formal Parameters
Purpose
Declarations/Instantiations
asks
Program Design Using Ada
Design Process
Informal Strategy
Ada Program Design Language
Develop Software Using Ada

NAME: Ada APPLICATION PROGRAMMER

DURATION: Four (4) Weeks

LOCATION: Keesler Technical Training Center or MTT at Users Site

OBJECTIVE: The course concentrates on the software engineer's use of the entire Ada language in designing and implementing Ada systems. The benefits of proper design techniques and good software engineering practices are emphasized. All aspects of the language are covered in depth. Over 50% of the course consists of hands-on programming practice. Object Oriented Design is taught and practiced in the course. Design and coding principles are brought together and applied in a comprehensive final project. Graduates will be able to design and code complex Ada systems and evaluate those written by others.

PREREQUISITES: This is not a basic programming course. Students are expected to have a working, fundamental knowledge of good programming concepts and recent experience in a high-order language. MTT requires a host site to have a validated Ada compiler with one terminal per student. Class size is limited to 8-12 students.

RECOMMENDED FOR: Programmers
Software Engineers

CCAF Credit: 5 Hrs

SYLLABUS: Block I: Fundamentals of Ada Software Engineering
Fundamentals of Ada Systems
Software Engineering
Language Concepts
Program Unit Structures
Ada Input/Output
Parameter Passing
Data Encapsulation
Scalar Types
Object Declarations
Discrete/Real Types
Control Structures
Sequential/Conditional/Iterative
Composite Types
Arrays/Records
Subprograms
Procedures/Functions

Block II Advanced Ada Software Engineering
Packages
Specification/Body
Context/Use Clause
Application
Exceptions
Defining/Raising/Handling

- Propagation
- Private Types
 - Abstract Data Types
 - Private/Limited Private Types
- Derived Types
 - Declaration/Usage
 - Derivable Operations
- Access Types
 - Declaration/Allocators
 - Unchecked Deallocation
- Generics
 - Declaration/Instantiation/Application
- Ada Input/Output
 - Text/Sequential/Direct I/O
- Tasks
 - Ada Tasking Model
 - Dependencies/Communication/Features
- Low-Level Features
 - Representation Clauses
 - Interrupts
 - Pragmas

- Block III Designing with Ada
 - Object Oriented Design
 - Software Design in Ada
 - Ada Program Design Language
 - Application Project

Bibliography:

1. Sammet, J., "Why Ada is Not Just Another Programming Language", *Communications of the ACM*, 29,8, pps. 722-733, Aug. 1986.
2. Booch, G., *Software Engineering with Ada*, Benjamin Cummings Publishing Company, Inc., Menlo Park, California, 1983.
3. Air Force Manual 50-2, *Instructional System Development*, 15 July 1986.

**"Integrating Ada into the University Curriculum:
Academia and Industry - Joint Responsibility"**

Kathleen Warner, Marshall University
Russell Plain and Kenneth Warner, Strictly Business

Introduction:

There is a well documented need for qualified Software Engineers able to work with Ada. This need far exceeds the current supply and demonstrates no signs of abatement. Industry needs personnel who have knowledge of the syntax, semantics, and uses of Ada and have been educated and trained within the Software Engineering discipline. In West Virginia and other states attempting to expand employment opportunities in the technical sector, the critical shortage of adequately trained personnel constrains development efforts.

Current undergraduate programs in Computer Science are insufficient to meet the demand for trained personnel in the Software Engineering industry. However, the demand must be met if the United States is to remain a leader in information technology. The education and training of individuals able to fill entry-level and advanced positions in the Software Engineering field must be understood to be a joint responsibility between the academic and industrial sectors.

Academic Requirements

As Computer Science has evolved as an academic discipline, the education of programmers, systems analysts, and other computer personnel has been primarily performed within colleges and universities. Academic programs in Computer Science necessarily serve the dual function of preparing students for entry into the workplace and alternately, for continued study on the graduate level. Academic institutions must maintain a careful balance in meeting these dual responsibilities.

Graduates seeking entry to a variety of professional fields in the software development industry must possess the necessary skills which meet basic industry needs. Preparation for entry into the workforce entails training in specific skill areas.

Students who will continue their education through advanced technical or graduate studies must have both depth and breadth in their undergraduate preparations. They must be knowledgeable over a broad range of topics encompassed within the discipline of Computer Science and possess the tools which are essential for continued research and development.

Academic Constraints

These dual responsibilities must concurrently meet internal curriculum guidelines and conform to minimal curriculum requirements defined by external accrediting boards. Standards and guidelines for academic programs are necessary but limit flexibility in modifying programs to meet changing conditions. Academic curriculum changes are accomplished slowly.

Planned modifications to curriculum may take as long as two years to implement. Approval for curriculum changes must be officially sanctioned through approval at the department, college, and university level. Students become aware of new programs and course offerings when the changes are officially published in the college catalog. Academic programs in Computer Science and Software Engineering are not evolving as rapidly as the Software Engineering industry.

Other constraints limit changes to existing academic programs. These include the availability of faculty and the cost of change. The hardware, software, and tools to support curriculum changes are often unavailable.

Industry Needs in Ada and Software Engineering

Existing industry needs for Software Engineers are not being met (Gerhardt, 1988) through undergraduate programs. An industry standard for preparing Software Engineers does not exist. In academic programs, most courses address software development issues as separate topics. The result may be graduates who have written many small (less than 1000 lines of code) programs which fail to exemplify the size and complexity of real world applications which can exceed one million lines of code. Future trends in software development are towards programs that are degrees of magnitude larger (Blumberg, et al, 1988). System intra-dependence and synthesis issues can often be the most complex aspect of real world implementation.

Addressing Deficiencies in Recently Hired Graduates

Industry must attempt to supplement the knowledge and skills of recently hired graduates to get them to a production level. Many different approaches are utilized to address these knowledge deficiencies.

Internal Training:

Many companies have internal training programs. These programs require an immense investment of resources and time. Trainees

are not productive until the vast majority of their training is completed. In addition, the tremendous costs often impact on the allocation of resources given to internal training, resulting in underfunded and overpaced programs. Since no two companies do internal training exactly the same way, inconsistencies in Software Engineering knowledge arise throughout the industry and even within a single corporation.

Consultants:

Corporations may attempt to quickly infuse technical proficiency by hiring consultants to provide expertise. For limited applications, such as training on a single package of software, consultants can be an effective solution. However, this approach is always very expensive and, unfortunately, not always effective.

Consultants, unfamiliar with the informal structure of an organization, will not provide training through osmosis. Software Engineers develop their skills through many hours of study, analysis and practice. It can take months to attain Software Engineering proficiency (Brownsword, 1988).

Remote Training:

Remote training is provided by outside vendors. Expensive on a per capita basis, this approach is not always effective. The training is brief and its application may not immediately follow. A corporation loses the Software Engineer's productivity for the duration of the course. Finally, courses are usually integrated to the vendor's environment and the results are not always transferable to the production environment.

On the Job Training:

Many corporations only provide informal training. Self taught students will have to do things the wrong way at least once. Constant project rescheduling and frequent program revisions to overcome ignorant oversights makes this the most costly and least effective training approach.

Night School:

Frequently, local colleges may provide an alternative source of training. However, there are many drawbacks. The available courses may be of inconsistent quality, class materials are not always relevant to the Software Engineer's application at work, and it can take months or even years to complete. Most importantly, this approach taxes the physical and mental stamina of full-time workers. Since this approach is often voluntary, total commitment and participation is rarely achieved.

Ignore the Problem:

Since most entry level graduates do not have sufficient Software Engineering knowledge, ignoring the problem can be disastrous. The size and complexity of present software development present challenges to even the best trained Software Engineers. Undertrained staff assigned to complex projects quickly become dissatisfied and unproductive. As a result, they may leave the project or company.

Training unprepared Computer Science graduates costs money and time. Very few companies are investing what it takes to develop well trained Software Engineers. When an organization has to train new employees, stress results for those being trained and others around them.

The Academic Response - New Courses and New Programs

The impetus for change in academic programs in the computing sciences is often external to academia. The realization that changes are necessary may result from dialog concerning programming practices. Dijkstra's 1968 letter to the editor of the Communications of the ACM, "Go To Statement Considered Harmful" spurred reaction but ultimately, made the Computer Science community aware of the need for structured, reliable, programs. Since the early 1970's, Computer Scientists and programmers have followed a body of programming methods and techniques called "structured programming".

Recognition of the "Software Crisis" directly led to the development and evolution of Software Engineering as a branch of Computer Science concerned with the techniques of producing and maintaining large software systems. The focus of attention has shifted from basic language concepts, systems and their implementation to the construction of systems from discrete program modules (Haberman, 1986, p.29).

Ada was developed and evolved in conjunction with the discipline of Software Engineering. Ada stands apart from other programming languages in its support of Software Engineering principles, its standardization and compiler validation, and its ability to create portable code (Sammatt 1986). The use of Ada can lead to the realization of the software development goals of readability, clarity, reliability, efficiency, modifiability, and portability of software products.

Sammatt (1986) observed that "one of the key aspects of Ada is its usefulness for new types of software education." She remarked that Ada has been successfully used in teaching

specialized courses in numerics, concurrent processing, data structures and can be successfully used as a first class in programming (1986).

Universities and industry alike are keenly aware that selected sectors of computer and software development industries will be directly affected by the existence of Ada in much the same way that sponsorship of COBOL influenced programming practices and course offerings over the previous twenty-five years. Ada language and Software Engineering classes are slowly being integrated into Computer Science curriculum offerings. Ada is often introduced in specialized courses such as Software Design or through a Software Engineering principles class (Burd, 1986).

New course offerings in the Ada programming language or Software Engineering principles may first be introduced as seminars, special topics, or experimental classes. The development of normal classes evolve from the refinement of these special topic classes.

Software Engineering at Marshall University

Programs of study in the computing sciences must be periodically reviewed to maintain currency with ACM and IEEE model curriculum recommendations. The Computer and Information Science department faculty have recently completed a program review for the Bachelor of Science program offered in Computer Science. The course content of existing classes was revised and additional courses were added to the program of study. The restructured curriculum stresses a Software Engineering approach from the first class in programming methodology.

Software Engineering was introduced as a separate class through a two semester sequence of special topic classes. The first class covered concepts and principles of Software Engineering and the Ada programming language. Packages, modules, data typing and generic units were the primary focus of study. Instantiation was taught through use of standard I/O procedures. The required texts included Sincovec and Weiner, "Software Engineering with Ada and Modula-2" and Booch, "Software Engineering with Ada", with DEC Vax/Ada language reference manuals used as lab supplements.

In the second special topics class, Software Engineering principles were emphasized and applied within the context of the Ada programming language. More attention was given to separate compilation units, exception handling, program libraries, and the concept of reusable code.

To create an appreciation and understanding of the usefulness of generic units, students wrote generic packages for the creation of familiar data structures. Other lab assignments included the completion of a simple Math package and a Statistical package. Package and procedure specifications were provided for these assignments.

Programming assignments were completed using the VAX/Ada compiler supported by WVNET. Watt, Wichmann, and Findlay, "Ada Language and Methodology" was used as a required text. Supplementary materials included Booch, "Software Engineering with Ada" and Vax/Ada language reference manuals.

Principles of Software Engineering evolved from these special topics classes and has been incorporated into the restructured CS curriculum. Between its inception and its implementation, the course has undergone a number of unexpected changes. In contrast to Sammett's recommendation that "... Ada has been and should be used as a vehicle for teaching software engineering principles in both academic and industrial settings" (1986, p. 129), the course description was modified to remove Ada as the designated language to allow instructors to choose from Ada, Modula-2 or C. In the revised curriculum, this class is now elective rather than required.

The original course description proposed strong prerequisites. Students could enroll only after completing a three semester sequence that included Pascal, Data Structures, Algorithms and Verification Techniques, Introduction to Language Processors, and Operating System Concepts. However, relaxed prerequisites could allow underprepared students to enroll. Thus, the large scale projects which cover the principles of software engineering may not be familiar to students.

Requirements and Training at Strictly Business

To insure a basic level of knowledge in recently hired graduates, Strictly Business Computer Systems addresses four aspects of Software Engineering training.

Basic Requirements:

Strictly Business hires new entry level Software Engineers who have completed at least a Bachelors degree in Computer Science or a related discipline. They must be fluent in multiple languages including Pascal (or Modula-2 and/or Ada). They must possess knowledge of operating systems, compilers, program design, and computing theory. A specialty area such as graphics, database design, or communications is essential. Finally, they should possess good speaking, writing and

self-management skills. Satisfying these stringent criteria, the employee is ready to begin training.

Establish Production Level Capabilities:

This three month process takes about 480 person-hours to complete and is divided into three cumulative, evolutionary steps to broaden the Software Engineer's mind and skills. The first segment addresses Ada syntax and examination of all aspects of the language. Exposure to Software Engineering fundamentals follows, reviewing Ada from a Software Engineering viewpoint (See Appendix A). The focus becomes what is to be done rather than how it will be done (Weiner and Sincovec, 1984). Finally, Software Engineering tools are discussed, including integrated programming support environments and computer aided Software Engineering design.

Rigorous Hands-On Quality Control:

Throughout the training, automated tutorials and lectures are supported with assignments of homework, outside readings, lab projects, and structured walkthroughs of developed software. In addition, verbal examinations help the instructor assess the rate of material absorption. Peer review and group problem solving sessions stimulate interest and serve as self-perpetuating quality controls.

Known Problems with this Approach:

Even a training approach of this depth has shortcomings. The duration of exposure to Ada and Software Engineering principles is not long enough. Yet, this three month training period is very expensive, particularly for small companies. Both trainees and trainer must make a strong personal and professional commitment to the training process. This aspect of the training is difficult. Not all trainees will succeed. Employees may be anxious that their positions depend on successful completion of the training. This training should be addressed within undergraduate Computer Science programs, so that companies hiring new graduates would only need to provide training to fine tune skills to their particular requirements.

The Team Effort

Only a team effort between academia and industry will provide a solution to the Software Engineering demands and requirements of the 1990's. This effort must involve continuous communication. College programs must be sensitive to the changing needs of the software development market, providing research, courseware and graduates that satisfy current and future demands. Businesses must provide direct feedback in response to the efforts of higher education. In this way, both universities and industry will become better informed. Some

suggestions to facilitate this process are offered. These opportunities for practical, real world experience with Software Engineering using Ada ensure that courseware satisfies academic requirements.

Team Teaching:

Marshall University CIS faculty and the Strictly Business Software Engineering Team have agreed to team teach a pilot offering of Principles of Software Engineering in the spring of 1990. The difficulty of teaching students how to "program in the large" with small scale, textbook examples will be countered with lectures and demonstrations from real world examples presented by practicing Software Engineers. Students benefit from actual examples of current industry applications. CIS faculty will maintain responsibility for assignments, tests, grading, and lecture quality.

Co-Op and Internship Programs:

During summer break or semester intersessions, students work as software engineers and could receive course credit. This option requires rigorous feedback mechanisms to insure uniformity in the work assigned and how the work is judged. The employer providing access to the Internship or Co-Op program must be involved in the grading process - the student should not receive a "Pass" grade for mere participation in these programs. Additionally, the student should be protected from internships that misuse the program's intention - corporations must provide assignments appropriate to Software Engineering training.

Scholarships and Grants:

Industry should donate funds to advance Software Engineering curriculum development. Funds might be used for development of Software Engineering courses, to obtain instructional materials, or to expand and enhance an existing program.

Recommendation

New directions in undergraduate programs preparing students for entry into the software development industry must be taken. We propose the development of undergraduate courseware which parallels the model curriculum of the Master's Degree program in Software Engineering at National University, California. Existing programs in Computer Science could be modified to include a Software Engineering specialization or universities might choose to implement Software Engineering as a separate, and distinct academic program. Academics and industry leaders should carefully examine and consider these proposals. Joint industry and academic efforts must start now - we are already late in starting.

Appendix A

A Sample Ada and Software Engineering Syllabus

- A. Types
 - 1. Unconstrained Types
 - 2. Unconstrained Record Types
 - 3. Attributes and Membership
 - 4. Real Types
- B. Statements
 - 1. If versus Case
 - 2. Exceptions
- C. Subprograms
 - 1. Parameter Passing Mechanisms
 - 2. Overloading
- D. Packages
 - 1. Specification versus Bodies
 - 2. Scope and Names
 - 3. Building an Abstraction
- E. Code Review
 - 1. Walkthrough
 - 2. Techniques
- F. Private Types
 - 1. Operations on Private Types
 - 2. Limited Types
 - 3. Building an Abstraction
- G. Generic Units
 - 1. Subprograms
 - 2. Formal Parameters
 - 3. Packages
 - 4. Instantiation
 - 5. Building a Generic
- H. Tasking
 - 1. Synchronous Communication
 - 2. Entries
 - 3. Accepts
 - 4. Types
- I. Program Library - Configuration Management
 - 1. Library Units
 - 2. Compilation Units
 - 3. Dependencies

- J. Ada Design
 - 1. Problem to Solution Mapping
 - 2. Minimizing Unit Dependencies
 - 3. Maximizing Reuse Potential
 - 4. Ada Shortcomings and Traps

- K. Design Review
 - 1. Walkthrough
 - 2. Quality Issues

- L. Round Table Discussion
 - 1. Course Summary
 - 2. Issue Discussion
 - 3. Further Readings - Extended Bibliography

References

- Blumberg, F., et al, "NASA Software Support Environment: Configuring An Environment for Ada Design", Ada Europe Proceedings, 1988, pp. 3-16.
- Brownsword, L., "Practical Methods for Introducing Software Engineering and Ada into an Actual Project", Ada Europe Proceedings, 1988, pp. 132-140.
- Booch, G., Software Engineering with Ada, Benjamin Cummings Publishing Company, Inc., Menlo Park, California, 1987.
- Burd, B., Teaching Ada to Beginning Programmers, Proceedings of the ACM, 1986.
- Gerhardt, M. "The Real Transition Problem or Don't Blame Ada", Tri-Ada '88 Proceedings, pp. 620-645.
- Haberman, A. N., Technological Advances in Software Engineering, Proceedings of the ACM, 1986.
- Sammet, J., "Why Ada is Not Just Another Programming Language", Communications of the ACM, 29,8, pp.772-733, Aug. 1986.
- Watt, D., B. Wichmann, and W. Findlay, Ada Language and Methodology, Prentice-Hall International, Englewood Cliffs, N.J., 1987.
- Weiner, R., and R. Sincovec, Software Engineering with Modula-2 and Ada, John Wiley and Sons, Inc, New York, 1984.

MAKING THE CASE FOR TASKING THROUGH COMPARATIVE STUDY OF CONCURRENT LANGUAGES

Michael B. Feldman
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052
(202) 994-5253
MFELDMAN@GWUSUN.GWU.EDU

Lt. Col. Frederick C. Hathorn
Information Systems Software Center
United States Army
Fort Belvoir, VA 22060
(703) 355-7025

INTRODUCTION

Support for concurrent programming, traditionally provided to the programmer by means of calls to operating system services, has received considerable attention from language designers in recent years. The goal of this effort has been to raise the level of abstraction of concurrent programming, providing ever more powerful language primitives, transferring responsibility for the details from the programmer to the compiler implementer. Indeed, what is happening now in concurrent programming echoes what happened in sequential "structured programming" perhaps fifteen years ago.

The study of concurrency has historically been a part of courses in operating systems. This view is a bit too narrow. In the current programming era, almost any interesting program embodies some aspect of concurrency, and so the issue should no longer be relegated to the relatively narrow application area of developing system kernels.

This paper describes a graduate course at The George Washington University entitled *Concurrency in Programming Languages*. Concurrent programming is taught from a comparative point of view, taking the student through the historical development through direct contact with languages implementing the various language primitives, and emphasizing language design and raising the level of abstraction rather than any particular applications.

Ada is the major unit of study; in a sense, the case is made for a tasking model at Ada's level through study of other languages with weaker tasking support.

Also presented in the paper is a brief description of Small-Ada, a personal-computer-based courseware package for the teaching of Ada tasking.

COURSE CONTENT

The course consists of lectures and language tutorials, a programming project in which four languages are compared, a student-selected term project, and an examination. Weights are 20% for the comparative project, 20% for the examination, 60% for the term project.

LECTURE MATERIAL

The lectures cover the basic formalisms, terminology, and application of concurrent programming, using as a starting point the excellent text by Ben-Ari [Ben-Ari 82]. Tutorials of one or several lectures each are given on the four languages covered comparatively, namely Co-Pascal, Modula-2, Ada, and Concurrent C.

Throughout the lectures, the emphasis is on the usefulness of concurrent programming as a framework for modeling the world, not just as an esoteric issue in operating systems or real-time applications. The world of modelled objects is a concurrent one. Historically, we have tended to try to model such a concurrent world with sequential programs, chiefly because our languages have grown up from the (von Neumann) computer toward the application domain, and not down from the domain to whatever computer is available for implementation. Recent concern for concurrency in languages has focused on better modelling of the world, together with the increased reliability that derives from abstraction and information hiding.

LANGUAGES STUDIED COMPARATIVELY

The comparative language study covers *Co-Pascal*, *Modula-2*, *Ada*, and *Concurrent C*. Each language introduces different notions of concurrent programming, as seen in the following brief survey. In each case a brief description of the relevant features is given. We also comment on the available implementations, emphasizing our commitment to make a wide range of compilers available, including some very inexpensive—or free—versions for students' use on their own personal computers.

Co-Pascal. Co-Pascal, an offshoot of Wirth's Pascal-S, is introduced in the Ben-Ari book. Source code for a compiler and P-code interpreter are given in the book appendix. Co-Pascal provides a very nice implementation of nondeterministic logical concurrency, but little in the way of structures for other abstraction, encapsulation, or inter-process communication.

- Co-Pascal uses COBEGIN and COEND to create and activate processes; a process is any Pascal-like procedure, including parameters; the same procedure can be spawned as multiple processes.

- Synchronization is implemented using signals with SEND and WAIT operations; no other communication or abstraction mechanisms are provided.
- The P-code interpreter "time-slices" process execution, with pseudo-time measured in pseudo-instructions executed. A quantum is a *random* number of instructions; the dispatcher schedules next process randomly. There is, within the limited capacity of the system, a nice nondeterminacy that illustrates empirically the justification for mutual exclusion.
- We use a public-domain IBM-PC implementation by Charles Schoening [Schoening 86], who ported compiler from VAX to Turbo Pascal as a Drew University project.
- Since source code is readily available, ports and enhancements make good term projects: we are developing a Macintosh version, and a Sun implementation has been done at another college.

Modula-2. Modula-2 is Niklaus Wirth's systems programming language of early '80's vintage [Wirth 1985, Ford 1985]. Designed essentially for workstations with multiple tasks but a single human user, Modula-2 embodies primitive (co-routine) support for concurrency, leaving it to the user to construct true process managers on top of this primitive support. Modula-2 provides good abstraction and encapsulation structures, but only low-level primitives for concurrency.

- Modula-2 provides good support for building systems of components (library modules with separately compiled interface and implementation files)
- There is usefully primitive support for logical concurrency (i.e. interleaved on a single processor): the pseudo-module "system" exports primitives for ~~creating PROCESSES~~ (which are just co-routines) and transferring between co-routines
- A parameterless procedure can be turned into a co-routine (perhaps *many* identical co-routines) by a call to NEWPROCESS.
- Some implementations provide for priorities and I/O interrupts, but no other scheduling, synchronization, or mutual exclusion is provided; this is precisely the virtue: students can experiment with building higher-level concurrency control and process managers.
- We illustrate several process managers: the one suggested by Wirth, a simpler one originally distributed with a now-defunct implementation and adapted by the author, and a very complete one incorporating time-slicing, developed at the University of Texas by Brumfield [Brumfield 87].

- Modula-2 is widely available at low cost: we give out a good \$35. shareware compiler (FST) for IBM PC, and a decent Macintosh compiler (MacMeth) with permission of ETH Zurich. We have reasonably good and inexpensive (or free) implementations for Vax-VMS (University of Hamburg), VM/CMS (Berlin) and Sun/Unix (Karlsruhe).

Ada. We assume that the reader is reasonably familiar with Ada; we include this brief summary so the reader will understand the Ada philosophy as we present it to students.

Designed as a general-purpose programming language for the complex defense-related systems of the '80's and '90's, Ada embodies, by design, very sophisticated mechanisms for concurrency, abstraction, and encapsulation [DoD 83, Gehani 84].

- The *package* provides good system-building capabilities: separately-compiled interface and implementation files, private types whose implementation is hidden from client programs, etc.
- Support for concurrency is intended to be high-level and independent of the operating system, using the extended rendezvous and "select" statement for synchronization and communication, and providing for timeouts, queued message-passing, and mutual exclusion without semaphores and signals.
- Processes are started by a mechanism similar to COBEGIN, and terminated by a fairly complex mechanism which tries to ensure that termination follows block structure and doesn't leave "orphaned" processes running.
- Ada is more standard than most languages, because of government-enforced compiler testing and validation; validated compilers are readily available for most computers, including two for the MS-DOS family for \$99.00.
- We use Meridian AdaVantage on IBM AT and PS-2 microcomputers. We also have TeleSoft and Verdix compilers on VAX/VMS, Sun/Unix, and IBM VM/CMS.
- We have developed a PC-based compiler as an adaptation of Co-Pascal, which supports the Ada tasking model with some educational features like user-selectable scheduling disciplines, enhanced task priority, process monitoring, etc. This courseware is presented in a bit more detail later in the paper.

Concurrent C. This is a C superset (also compatible with C++) developed at AT&T Bell Labs (Murray Hill) by Narain Gehani and colleagues [Gehani 89].

- Concurrent C consists of a preprocessor generating standard C (or C++), together with a run-time system supporting the process control.

- The Ada concurrency model is essentially grafted onto C, and extended to allow dynamic priority-setting, priority queues in addition to FIFO for message-passing, and, recently, asynchronous message passing in addition to the Ada synchronous model.
- Concurrent C serves as a good vehicle for studying what an improved Ada could look like, since most of Gehani's changes could be added upward-compatibly to Ada.
- The system is currently available to universities through Bell Labs: Sun and AT&T 3B implementations are distributed.

To summarize, this set of four languages serves as a broad survey of concurrency features. With the exception of Concurrent C, all languages are readily available for use on personal computers, so that students with their own machines can use them conveniently, and also continue with the languages after the course is over.

With the growing interest in true parallel processors, we have begun to incorporate some material on the occam language into the course as well.

A PROGRAMMING PROJECT IN COMPARATIVE LANGUAGES

The students are required to do a comparative exercise, to familiarize themselves with the languages under study, and to have some close-up experience with their concurrency and encapsulation mechanisms. Two important aspects of the comparative exercise are *code modification* and *algorithm animation*.

Code Modification. Unless they have had jobs in industry, students often get little experience in re-using other peoples' code, perhaps adapting it to new uses; they need this experience. First, in practice much program "maintenance" (read enhancement) is done in industry; second, programs are too often — in school and industry as well — written from scratch for each new application, with little attention paid to development of rich libraries and re-usability of programs.

In this project series, students write relatively little code *ex nihilo*. They are given listings and machine-readable files of pre-existing modules and directed to use them, perhaps after some enhancement work. Several sort programs, a terminal driver, a window manager, and a task dispatcher are all adapted, or translated from language to language, during the course of the project.

Code adaptation fosters a positive attitude toward re-use; one builds on the work of others instead of competing with it or re-inventing it.

Algorithm Animation. Projects like that at Brown University [Brown 85] are developing schemes for dynamic algorithm visualization on high-resolution workstations. Animation is *fun* and helps to hold students' interest, and — within limits — it can be done "cheaply" with 24x80 "dumb terminals." The advantage of the cheap approach is that it can be done portably.

A very good way to understand the interaction of concurrent programs is to have them dynamically display — animate — their state. We have encouraged this idea through the vehicle of animated sort algorithms. Sorts are easy to work with: students understand them well and can therefore pay attention to the animation and the concurrency rather than to the algorithm being animated.

Racing Sorts. Four files are provided: three are procedures, each implementing one well-known sorting algorithm for arrays of up to 64 characters, for example BubbleSort, LinearInsertionSort, HeapSort. The fourth file contains a module for a miniature terminal driver, exporting ClearScreen and SetCursorAt operations. A demonstration file is distributed, in which a single sort procedure displays its array of characters on a single row of the display, then posts the changes to the array as the sort proceeds.

The requirements of the project are—for each of the four languages—to turn each of the three sort procedures into a process, then start the three processes in a simulation of a "race" to completion. The screen is a shared resource (at least conceptually), to which all sorts must write; further, ANSI terminal control requires several characters of overhead to position the cursor. In order to guarantee that a "transaction" to the screen is completed successfully, then, the synchronization and communication primitives of each language must be used to build a monitor or other mutual-exclusion mechanism.

A typical initial screen display is given in Figure 1. Execution of the program gives the visual effect of animation as values are interchanged. Indeed, depending on the implementation, the animation may be too rapid for the eye to perceive, so the student must slow the action down with a delay of some kind.

For the student wishing to experiment a bit with methods of displaying text in different windows, the source code for a simple window manager is distributed. An example of the window manager in operation is shown in Figure 2. This window manager does not have any mutual-exclusion code; the student must build it in.

```

SORT RACE

BUBBLE SORT
aZAzbYBycXCxdWDweVEvfUFugTtGhSHsiRiR

LINEAR INSERTION
aZAzbYBycXCxdWDweVEvfUFugTtGhSHsiRiR

HEAPSORT
aZAzbYBycXCxdWDweVEvfUFugTtGhSHsiRiR

PRESS RETURN TO BEGIN THE RACE

```

Figure 1. Initial Screen for Sort Race.

```

I am now wr
iting in th
is window.
Notice how
the text wr
aps

the quick
brown fox

Here is the text appearing i
n the third window.

```

Figure 2. Example of a simple window manager.

TERM PROJECT

An important part of the course is the term project, which the student selects, proposes, implements, and reports on in a public (class-wide) forum. Following is a summary of the project structure.

- Projects are generally individual, but may be done with small team or "double-credited" to two project courses.
- Oral proposals (5 mins.) are given in class; peer review is often interesting and useful.
- Progress reports (15 mins.) are given during the last two classes.
- Projects are due at the final exam - sometimes "incompletes" allow extra time for completion of particularly good projects

Typical Projects. Here are a few examples of interesting student projects:

- Develop an interesting process manager for Modula-2; demonstrate with interesting example.
- Simulate a graphical monitor for a multi-computer network using Meridian Ada and PC graphics.
- Build an asynchronous mouse interface to Meridian Ada and demonstrate with an interesting example.
- Develop a video game with game actors as processes.
- Interface the Sun graphics library to Ada, making an Ada-callable library; develop an animated cartoon of dining philosophers (2-person project).
- Build a new high-level PC graphics library and use it for an interesting multi-process animation in Modula-2 (double credit with graphics course).
- Develop a multi-process implementation of an AI-oriented game-winning strategy (double credit with AI course).
- Compare and contrast recursive and concurrent implementations of divide-and-conquer algorithms.
- Study parallelism; benchmark serial and parallel versions of an algorithm on a supercomputer.

Small-Ada: COURSEWARE FOR TEACHING Ada TASKING

Tasking is a particularly interesting aspect of Ada; it is also less well-understood than most of the sequential features of the language. In addition, the Ada Language Reference Manual leaves many tasking details unspecified, preferring to delegate a

great deal to the implementor. Important examples of implementor choices are the number of task priorities, the task-scheduling strategy (e.g. presence or absence of time-slicing) and the details of the arbitrary selection called for in the `select` statement.

The typical commercial Ada compiler, including those we have available, embodies a set of design choices in the tasking model which are not always well-documented and are, in any case, not alterable by the user. To foster effective study of the tasking model, with specific reference to the portability of programs in the presence of different implementor choices in the tasking model, it is helpful to have available an implementation whose tasking model can be controlled at compilation time, or at least whose run time system is accessible so that it can be re-coded if necessary.

To provide the wherewithal for comparative study of tasking implementations, we have developed a system for personal computers called Small-Ada. Coded in Turbo Pascal, Small-Ada supports an approximately full Ada tasking model but relatively little of the sequential language (e.g. packages, generics, access types, etc., are not supported).

The compiler produces a variant of P-code, which is executed under control of an interpreter. Since the entire system is relatively simple (we have emphasized the tasking model without carrying the translator baggage of full sequential Ada) it can be modified with relative ease as a student project. Since it is written in Turbo Pascal, students can take it to their personal computers and work in quiet comfort.

As the system has evolved, we have built in an increasing number of user-selectable tasking options. Currently the user can choose time-slicing or not (and if so, the size of the quantum); random vs. deterministic selection of the next-to-be-scheduled task; standard Ada static priorities vs. dynamic priorities (as is under consideration in the Ada9x project); implementation of the `select` statement including priority inheritance. User choices are implemented as pragmas, to maintain standard Ada syntax.

A recent enhancement incorporates a considerable degree of system monitoring, with each task's state, and parts of its source code, shown in its own window. Implementing the target machine as a pseudo-machine has made it possible to do interesting task-state monitoring without influencing the tasks' timing, since we can "stop the clock" while the instrumentation does its work.

We have also developed a Macintosh version of this system. Plans for the future are to improve the user interfaces, monitoring capabilities, and tasking options of both systems. The IBM-PC version is available from the authors; the Mac not yet.

CONCLUSIONS AND CLOSING COMMENTS.

In university courses comparative study is a traditional and very effective way of organizing knowledge. In the academic setting we believe very strongly that the best way to understand a design is to compare it with alternative designs; this comparative approach is also a classical engineering methodology. Computer science students are best served when they have been exposed to alternative strategies for the solution of a problem, including, in the present case, alternative language-design models.

We have found that learning the languages, *per se*, is not a large problem. Students with reasonable Pascal or C background can learn the rudiments of several derivative languages in a single course, given a "code modification" approach and care and guidance from the teacher.

Students emerge from CSci 358 with an understanding of the strengths and weaknesses of Ada's high-level tasking model compared with other models, and for those working in the Ada-related industry this is an important contribution to their professionalism.

REFERENCES

- [Ben-Ari 82] Ben-Ari, M. *Principles of Concurrent Programming*, Prentice-Hall, 1982.
- [Brumfield 87] Brumfield, J. *A Modula-2 Process Manager*. unpublished.
- [Brown 85] Brown, M.H., and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, Vol. 2, No. 1, January 1985, pp. 28-39.
- [DoD 83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A, 1983.
- [Ford 85] Ford, G.A., and R.S. Wiener, *Modula-2: A Software Development Approach*. John Wiley and Sons, 1985.
- [Gehani 84] Gehani, N. *Ada: Concurrent Programming*, Prentice-Hall, 1984.
- [Gehani 89] Gehani, N. and W.D. Roome. *Concurrent C*. Silicon Press, 1989.
- [Schoening 86] Schoening, C.B. "Concurrent Programming in Co-Pascal," *Computer Language*, September 1986, p. 32-37.
- [Wirth 85] Wirth, N. *Programming in Modula-2*, 3rd corrected edition. Springer Verlag, 1985.

Teaching the Ada Tasking Model to Experienced Programmers: Lessons Learned

John P. J. Kelly Susan C. Murphy
Dept. of Electrical & Computer Engineering
University of California, Santa Barbara, USA

1. Introduction

What can go wrong when 18 graduate students implement their first Ada program? What if the application involves extensive use of concurrent communicating tasks? And what if those programs are generated from multiple formal specifications? We learned many lessons during an experiment performed at UCSB to investigate software engineering techniques for distributed systems with very high reliability requirements. One important motivation for the project was to investigate the problems encountered in building complex concurrent processing applications in Ada. We are involved in building fault-tolerant software for dependable systems and so are particularly interested in understanding the faults that occur in complex systems: the types of faults, their underlying cause, and their effect on an operational system [KEL 88]. By understanding the faults that commonly occur, we can better train our programmers to reduce faults and also build software systems that can tolerate the faults that remain. Since specification defects have been pinpointed as a major source of faults [AVK 84], we are evaluating the benefits and difficulties encountered in implementing a concurrent application from formal specifications. Our analysis addresses several issues:

- Use of Diverse Formal Specifications
- Use of Ada for Distributed Applications
- Analysis of Specification, Design, and Implementation Faults

2. The Diverse Protocol Specification Experiment

2.1 The Application

In our experiment, multiple independent versions of a distributed application were implemented in Ada from three diverse specifications written in different formal specification languages. The application is a communication protocol based on the Open Systems Interconnection (OSI) layered-model adopted by the ISO [ISO 84]. A communication protocol describes a set of rules for interaction between end-users on different computing systems that allows them to exchange information. The OSI model provides a reference for the design of *standardized* communication protocols. The specification languages, Estelle [ISO 87], LOTOS [ISO 87a], and SDL [CCI 88], represent international standards for specifying OSI communication protocols [ISO 84]. The three specifications were independently developed by experts. They describe an OSI Transport Protocol which is a relatively complex application [ISO 88]. The transport protocol allows different computing systems to exchange information by synchronizing and controlling the transfer of data between two machines. Such a protocol involves a significant amount of concurrency and synchronization among cooperating tasks. In the experiment, the protocol implementations were developed independently and then underwent a carefully controlled validation process involving extensive testing and debugging. We used an automated test procedure called *back-to-back testing* in which the outputs were compared to detect errors [KM 89a]. The details of our research work, the experimental paradigm and test environment, and the results have been described elsewhere [KM 89b].

2.2 The Programmers

The programmers were graduate students in Computer Engineering enrolled in a seminar class. All were experienced Pascal or C programmers, although only one student had experience using Ada to implement a concurrent program. None of the programmers had extensive knowledge of communication protocols or of the formal specification languages. So far, the experiment has comprised two stages. During the first pilot stage, performed last year in which six implementations were completed and tested, we focused on improving our methodology, correcting specification defects discovered by the programmers, and analyzing the programming errors. The second stage of the experiment has just been completed with nine implementations.

2.3 Initial Results

We learned a great deal during the pilot stage, not only from the programmers' questions and problems, but also from an analysis of the faults discovered in their programs. Many of the problems were related to defects in the specifications, but there were also many difficulties related to the semantics of the Ada tasking model and the complexity of concurrent programming. The programmers were not able to comprehend all of the possible parallelism in the system and did not appreciate (at least initially) the problems associated with concurrent execution of the multiple tasks. Of course, just being aware of the need to analyze the concurrency in a system is not enough; determining all of the actions that can occur in parallel is usually very difficult. In general, the most difficult faults to find and correct were those related to synchronization, typically caused by incorrect assumptions about what was going on elsewhere in the system.

2.3.1 The Ada Tasking Model

In Ada, tasks communicate via the *rendezvous* in which the sending task issues an entry call to the receiving task and the receiving task accepts the call for that entry. The first task arriving at the rendezvous point is suspended until the other task arrives. The rendezvous represents a *synchronous* communication model in which the sender is not released until the receiver has accepted and processed the message. A different model is represented in *asynchronous* communication in which the sender is allowed to proceed immediately and the messages queued for delivery to an autonomous receiver. Since both communication models were important in our application, the lack of asynchronous facilities in Ada meant that the programmers had to devise their own methods of providing asynchronous (i.e., non-blocking) inter-task communication. A simple mechanism for achieving asynchronous communication between two tasks is to introduce an 'agent' task between them (also called 'mailboxes') [BLW 87]. While this approach works well in some cases, in the protocol implementations the inter-task interactions were complex and asynchrony more difficult to achieve. The programmers' misuse of the synchronization mechanism in the form of the rendezvous resulted in *deadlock*, a common system failure in the experiment. Deadlock occurs when synchronizing tasks are suspended waiting for a rendezvous which will never occur. In practice, this was a potential problem whenever a task had to function as both a sender and a receiver, thereby mixing entry calls with **accept** statements. Since the Ada rendezvous blocks the caller until the receiver is ready to accept the call, deadlock can ensue in such cases.

Although our programmers were warned about the possibilities of deadlock, few appreciated the problem until their programs actually deadlocked in operation and they were forced to find the offending fault. The deadlock errors were the most difficult to detect and correct and also had the most severe consequences since system failure was the usual result. Trying to correct program errors related to deadlock was particularly difficult late in the development process (for example, during testing) since correction often required extensive redesign. Several problem areas for the programmers included:

- When the transmitting task sent a message over the communication medium, it set a timer (usually implemented as a separate task) to signal time-outs for receipt of acknowledgements from the receiving site. If the timer task tried to signal a time-out to the transmitter task at the same time the transmitter task was trying to send a reset signal to the timer task, deadlock occurred
- When the buffers in the transmitting task were filled, the transmitter blocked further transfer of data. If the programmer was not careful the transmitter task was no longer open to accept acknowledgements from the receiver (which was the only means to free buffer space!), resulting in a deadlock
- When the transmitter task sent data, it had to set a timer at the same time. If the data was sent before the timer was set, sensitive timing failures occurred if an acknowledgement was received and cancellation of a possibly non-existent timer was attempted

Probably the most frustrating problems in the first stage of the experiment were caused by compiler bugs that forced programmers to redesign their semantically correct code to avoid the faults in the compiler. In the second stage, we switched to a different, more mature compiler which worked (almost) all of the time.

2.3.2 Mapping Formal Specifications into Ada

In analyzing the deadlock faults in the programs from the first stage, we discovered that the problems often stemmed from trying to map the formal specification, written in a language with its own semantics for communicating processes, into an implementation in Ada, a language with its own (and different) semantic model. Although we knew that the specification languages have their own concurrency models and that the Ada language has its own semantics for concurrency, we did not realize how the often subtle mismatch could cause programming errors. In fact, none of the specifications was concerned with the potential deadlock problems that can arise in an implementation. However, the specification languages were very implementation-oriented and the protocol specifications were, in fact, *design specifications* of the system. This misled the programmers who often assumed the specification design could (and should) be mapped directly into Ada. Although the programmers tried, none of the specifications could be mapped directly into Ada because of semantic differences, particularly with respect to the synchronization in interprocess communication. In the first stage in particular, the programmers had great difficulty in understanding what went wrong. The design documents the programmers submitted early in the design phase differed from their final design documents, and most of the redesigned sections of their documents had to do with avoiding deadlock problems.

The programmers complained that the specifications were not helpful for understanding the dynamic concurrency in the system and, furthermore, that the specifications' partitioning of the system into multiple processes encouraged the use of tasks more than was necessary for the application. For example, the specifications assumed the existence of an autonomous timer for signaling time-outs in the protocol, implying the need for a separate task to implement the timer. In fact, this was not the case, a separate timer task was not the only way to implement a time-out, and yet all but one programmer implemented the timers as separate (usually multiple) tasks. The timers were a common source of deadlock. In general, methods devised for avoiding deadlock in the protocol implementations usually involved the use of timed 'polling' loops in existing tasks or the creation of new tasks (which may also have polling loops in them) to handle the communication causing the deadlock. Of course, both solutions have their drawback. With timed polling, program execution is very sensitive to the delay time in the polling loop, degrading performance and making it difficult to predict the expected behavior. On the other hand, when the number of tasks in a system is increased, performance tends to decrease, correctness becomes more difficult to assure, and deadlock is harder to avoid.

An analysis of the programmers' difficulties gave us some surprising insights into the problems of implementing communication protocols in Ada and pinpointed errors in previously published work in the use of Ada for implementing communication protocols [CDG 88]. All three formal specifications had numerous defects which caused implementation errors. It became clear that programmers are not the only ones who have difficulty with concurrent applications -- specification writers and compiler writers have an equally difficult time. Distributed software engineering in Ada poses many new and challenging problems for software development, not only in the implementation phase, but also in specifying the application requirements and in testing the final implementations.

3. Teaching Ada

During the experiment, we provided four weeks of Ada training, emphasizing Ada's special features such as packages, generics, exceptions, compilation, and tasks. In presenting tasks, we focused on the semantics of the tasking model, its constructs for synchronization and communication, and the potential for deadlock between communicating tasks. Our teaching approach was different from what we would have taken if the goal had been to teach software engineering using Ada; in fact, since the goal was to study the faults experienced programmers make when working independently, we did not want to dilute the diversity in results by presenting a particular approach to Ada software development.

3.1 Lessons Learned

Learning from our initial experience, our Ada training efforts during the second stage were geared to getting the message across early about potential problems relating to synchronization and concurrency and to mapping formal specifications into Ada. We knew that just explaining the problem would not be enough; allowing the programmers to discover it themselves with a 'hands-on' practice exercise was an important reinforcement to the training sessions. Thus, in the second stage, we assigned the programmers a simple specification of two communicating processes to be implemented in Ada. The example protocol specification, which *seems* to have a straightforward implementation, will result in deadlock if the specification is mapped directly into Ada without careful consideration being given to the different semantics of the languages. The programmers tried to do just that and had deadlock. Several of the programmers insisted that the protocol was wrong. Implementing the example in Ada gave the programmers a clear understanding of the problems they faced and, as a consequence, they were better prepared for their more complex protocol specification. As a result, in the second stage the programmers had considered deadlock avoidance in their initial protocol design. However, analysis of the faults in their implementations indicates that while part of the problem was solved with the practice exercise, a few of the programmers still had difficulty understanding the task interactions and the finer points of the rendezvous.

From our experience, we suggest that a training program include the use of a dynamic model, (we prefer a petri net model that allows dynamic simulation of the task interaction behavior), to explain the operational semantics of the rendezvous construct. Our analysis makes it clear that the programmer must have an understanding, not only of the basic semantics of the tasking model, but also of its operational effect in a concurrent program. More time should also be spent explaining the run-time implementation of tasks and the rendezvous. Just as recursion is made more clear by exposing the underlying stack implementation model, so too can tasking be explained by describing its implementation.

The remainder of this paper presents the sample specification, several programmers' solutions and their errors. This presentation provides insight into the problems associated with mapping a formal specification into Ada and the difficulties involved in comprehending a system (even a simple one)

with multiple task interaction, particularly when the communicating tasks must be prepared to send and receive messages.

4. Training with a Practice Exercise

4.1 The Sample Specification

The goals of this sample training exercise include familiarizing the programmer with the problems that arise in developing concurrent applications and in mapping a formal design specification into Ada. The example assumes that there are two communicating tasks in the system, each following the same behavior specification. The two tasks periodically send and receive messages from each other, verifying that each is still alive. When a task receives a message, it must respond by immediately sending an acknowledgement to the sending task. If one task has not heard from the other (i.e., either received a message or an acknowledgement) within the past 20 seconds, it sends a message. After sending a message, the sending task should receive an acknowledgement within 3 seconds; if the receiving task does not communicate within 3 seconds, it is assumed to be dead and the sending task should also terminate. The programmer must implement and test this system of communicating tasks according to the formal specification. We have included two sample specifications of these informal requirements: one a simple state transition diagram with no formal specification language used (Fig. 1) and the other written in SDL, a formal description technique for communication protocols used in our experiment (Fig. 2). The SDL specification illustrates how an explicit timer is used to specify the time-out requirement. This example emphasizes experience with programming from formal design specifications, the use of real-time constraints, mutually communicating tasks, and potential deadlock. It also requires that the programmer design a test driver and test cases to test the communication behavior to ensure conformance with the specification.

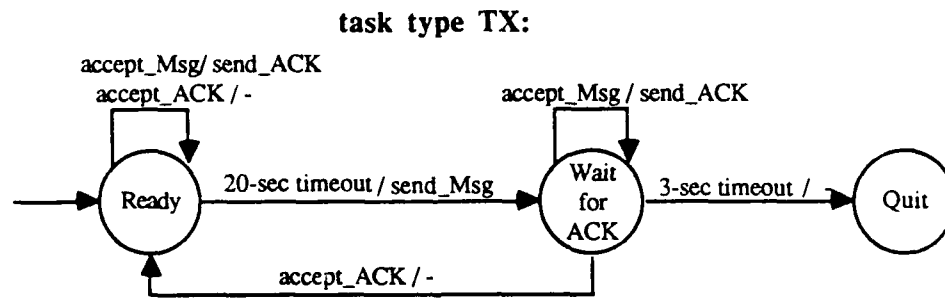


Figure 1. State Transition Diagram for Communicating Task

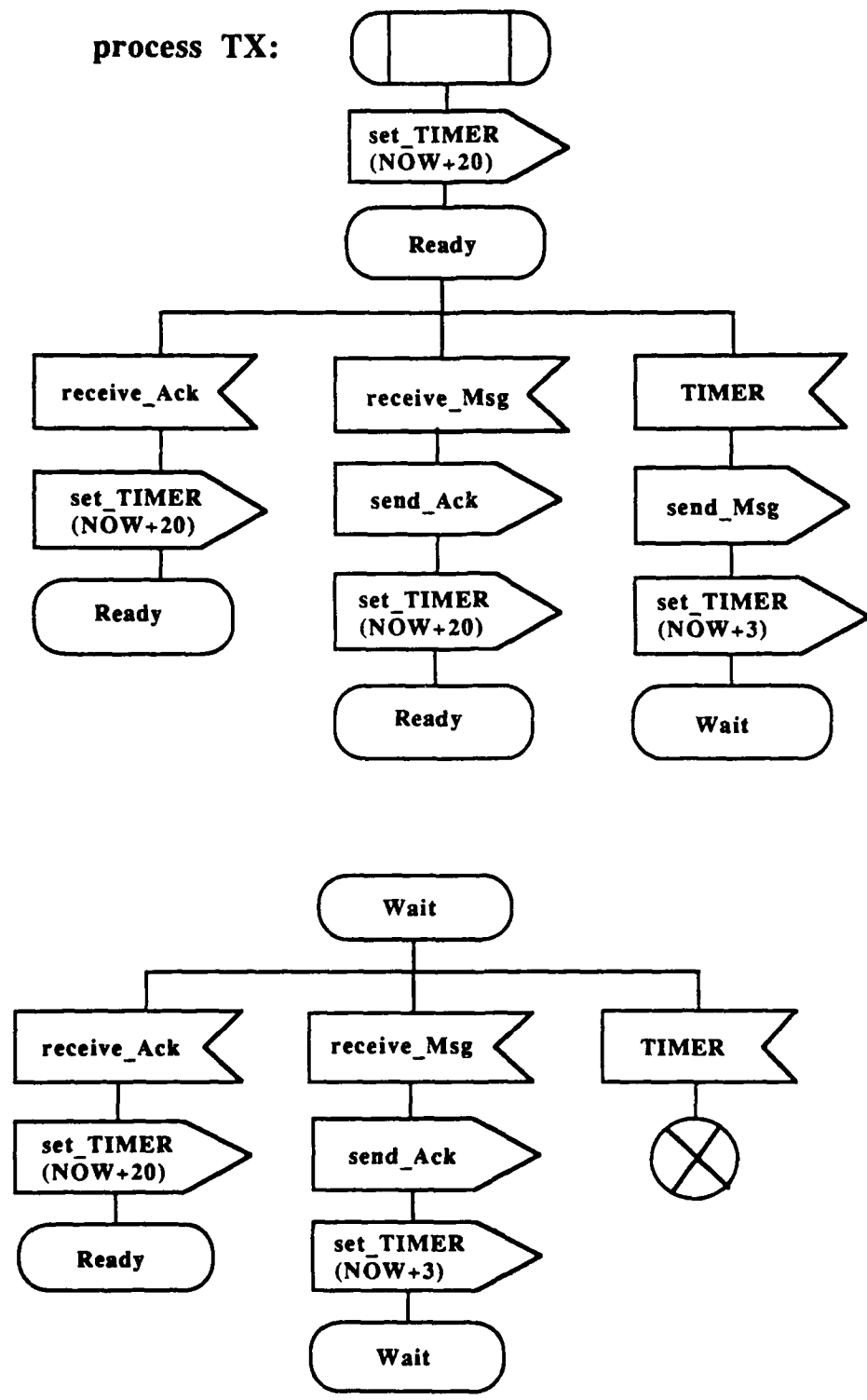


Figure 2. SDL Specification of Communicating Task

4.2 Programmers' Solutions

The following represents several programmers' solutions to the practice specifications. The two communicating tasks are named TX_A and TX_B. Only sketches of the solutions are provided.

- Solution 1: This proposed solution was attempted initially by most programmers and represents a close mapping to the state transition diagram (Fig. 1). It results in deadlock quickly and led some programmers to assume that the specification must be in error. Note that *delay* statements are used to implement the time-outs instead of a separate timer task. Separate timer tasks only exacerbates the deadlock problem (as illustrated in solution 2 below).

```
task body TX_A is
  type STATE is (READY, WAIT);
  STATE : STATETYPE := READY;
  type MSG_TYPE is ...;
  begin
  loop
    select
      accept MSG (M : in MSG_TYPE) do
        -- read & process M
      end accept;
      TX_B.ACK;
    or
      accept ACK;
      STATE := READY;
    or
      when STATE = READY =>
        delay 20.0;
        -- prepare message M for delivery
        TX_B.MSG (M);
        STATE := WAIT;
    or
      when STATE = WAIT =>
        delay 3.0;
        exit;
    end select;
  end loop;
end TX_A;
```

What goes wrong? The deadlock arises because the other task in the system, TX_B, will eventually try to send a message to TX_A at about the same time TX_A tries to send a message to TX_B. Each programmer came to realize quickly that deadlock avoidance was up to them and was not handled in the specification. The SDL specification language assumes an asynchronous communication model in which the sender of a message is not blocked. However, problems are not eliminated by simply switching to a programming language with an asynchronous model. Asynchrony leads to other timing problems: task B may not get around to accepting a message from A in time to send an acknowledgement within the 3 second time-out and A may (incorrectly) assume that B has died and thus terminate itself. In an asynchronous model, assumptions have to be made about the maximum time necessary for message receipt.

- Solution 2: This solution is similar to the first case except that a separate timer task is used to implement the time-out. This solution represents a close mapping from the SDL specification (Fig. 2).

```

task body TIMER is
  TIME_DELAY: DURATION;
begin
loop
  select
    accept SET (DELAY_PERIOD: DURATION) do
      TIME_DELAY := DELAY_PERIOD;
    end SET;
  or
    delay TIME_DELAY;
    TX_A.TIMEOUT;
  end select;
end loop;
end TIMER;

```

```

task body TX_A is
  type STATE is (READY, WAIT);
  STATE : STATETYPE := READY;
  type MSG_TYPE is ...;
  DELAY_PERIOD : DURATION := 20.0;
begin
  TIMER.SET (DELAY_PERIOD);
loop
  select
    accept MSG (M : in MSG_TYPE) do
      -- read & process M
    end accept;
    TX_B.ACK;
    TIMER.SET (DELAY_PERIOD);
  or
    accept ACK;
    DELAY_PERIOD := 20.0;
    TIMER.SET (DELAY_PERIOD);
    STATE := READY;
  or
    when STATE = READY =>
    accept TIMEOUT;
    if STATE = WAIT then
      exit;
    end if;
    -- prepare message M for delivery
    TX_B.MSG (M);
    DELAY_PERIOD := 3.0;
    TIMER.SET (DELAY_PERIOD);
    STATE := WAIT;
  end select;
end loop;
end TX_A;

```

A timer task similar to this had been proposed for use in communication protocols in Ada [CDG 86]. A similar timer task was tried by several of our programmers in their protocol implementations but with no success. The failure was caused by deadlock which ensues whenever the TIMER task attempts to rendezvous with the TX_A task (because it has timed out

and is executing the TIMEOUT call) at the same time the TX_A task is trying to rendezvous with the TIMER at the SET entry (because it has heard from TX_B and is trying to reset the timer). Since each task is waiting at different rendezvous points for the other, deadlock will occur.

- Solution 3: The first example of a solution for deadlock avoidance is to provide *message forwarding* tasks to implement an asynchronous (non-blocking) send. The forwarding task accepts a message from the TX_A task and passes it on to TX_B, avoiding the blocking of TX_A. This solution results in two additional tasks; a forwarding task for each of the two TX tasks. The TX_A task above would be modified slightly to send messages to TX_B via an entry call to the TX_A_FORWARD task which accepts messages and forwards them on to TX_B as follows:

```
task body TX_A_FORWARD is
MESSAGE : MSG_TYPE;
begin
loop
select
accept SEND_MSG (M : in MSG_TYPE) do
MESSAGE := M;
end accept;
TX_B.MSG (MESSAGE);
or
accept SEND_ACK;
TX_B.ACK;
end select;
end loop;
end TX_A_FORWARD;
```

The duplication of tasks for message transmittal will lead to substantial overhead in a more complex system. Also this example's restrictions on the sending of acknowledgements and messages (only one message and one acknowledgement is sent in a 20 second interval) makes the solution simpler than if transmittal of multiple messages and acknowledgements was possible (as in the transport protocol).

- Solution 4: In this solution, the programmer avoided deadlock by a timed selective polling approach. The implementation is similar to the first solution except that additional state variables are added to determine if the task was blocked sending a message. When a task wants to send a message or an acknowledgement, then a new state variable, 'TRYING_TO_SEND_MSG' or 'TRYING_TO_SEND_ACK', is set to true and a short delay time for polling is assigned. The task tries to send the message or acknowledgement with a select/delay alternative as follows:

```

task body TX_A is
  POLL_TIME : CONSTANT DURATION := ...; -- small delay time
  -- ... declarations similar to solution 1
  loop
    select
      accept ...
      .
      .
      -- .similar to solution 1 with the following additional alternatives
    or
      when TRYING_TO_SEND_MSG =>
        delay POLL_TIME;
        select
          TX_B.MSG (M);
          STATE := WAIT;
          TRYING_TO_SEND_MSG := FALSE;
        or
          delay POLL_TIME;
        end select;
    or
      when TRYING_TO_SEND_ACK =>
        delay POLL_TIME;
        select
          TX_B.ACK;
          TRYING_TO_SEND_ACK := FALSE;
        or
          delay POLL_TIME;
        end select;
    end select;
  end loop;
  ...

```

Situations such as this example, in which a task needs to be open to make entry calls and to accept calls from other tasks, leads to polling as a solution. The polling approach is expensive, requiring frequent execution of the `select` alternative. Of course the most natural approach requires the `select` statement in Ada to allow both entry calls and `accept` statements (of course, this is not an option). This solution represents an optimistic approach in that it can be efficient if the receiver task is usually ready to accept the call and the need to poll is rare.

- Solution 5: In this solution, the programmer avoided deadlock by implementing a single buffer task which acted as a passive depository (or mailbox) for all messages. The buffer task acts as a synchronizing agent for the TX tasks, accepting calls from the two TX tasks to add messages to the buffer or remove them from the buffer. The TX tasks only make entry calls to the buffer task and do not contain any accept statements themselves. While this approach avoids many difficulties, it isn't the most natural approach since it requires that the TX tasks reverse their roles, with each making entry calls to receive messages. This solution represents a pessimistic approach which works more efficiently if the receiver task is often busy and the buffering of messages frees the sender task from explicit polling (as in solution 4 above). The body of task TX_A is sketched below. The buffer task is not shown; it simply accepts calls from TX tasks with requests to either enqueue or dequeue messages.

```

task body TX_A is
  type STATE is (READY, WAIT);
  STATE : STATETYPE := READY;
  DELAY_PERIOD : DURATION := 20.0;
  .....
begin
loop
  select
    BUFFER.A_GET (KIND:M_KIND; M : MSG_TYPE);
    -- retrieve message M from buffer (if buffer not empty)
    -- process and take action on M
    .
  if KIND = ACK then
    DELAY_PERIOD := 20.0;
    STATE := READY;
  elsif KIND = MSG then
    -- deposit acknowledgement into buffer for later retrieval by TX_B
    BUFFER.A_SEND_ACK;
  end if;
or
  delay DELAY_PERIOD;
  if STATE = WAIT then
    exit;
  end if;
  -- prepare message M for delivery
  -- deposit message M in buffer for later retrieval by TX_B
  BUFFER.A_SEND_MSG (M);
  DELAY_PERIOD := 3.0;
  STATE := WAIT;
end select;
end loop;
.....

```

- Solution 6: This is really not a solution (it didn't work), but it is interesting in understanding the error in logic. Several programmers used this approach, based on the use of a *semaphore*. Typically, semaphores are used for allowing two processes to update shared data without interference or for allowing one process to block itself to wait for a certain event and then to be awakened by another process when the event occurs. The programmers assumed that if each TX task were to use a semaphore before and after an entry call to the other TX task that they would be guaranteed that only one process would be trying to make an entry call at the same time and deadlock would not occur.

```

task body SEMO is -- task implementing set/reset of semaphore variable
  IS_RESET : BOOLEAN := TRUE;
begin
  loop
    select
      when IS_RESET =>
        accept SET do
          IS_RESET := FALSE;
        end SET;
      or
        accept RESET do
          IS_RESET := TRUE;
        end RESET;
    end select;
  end loop;
end SEMO;

```

```

task body TX_A is
.
.
-- similar to solution 1
loop
  select
    accept MSG (M : in MSG_TYPE) do
      -- read & process M
    end accept;
    SEMO.SET;
    TX_B.ACK;
    SEMO.RESET;
  or
    accept ACK;
    STATE := READY;
  or
    when STATE = READY =>
      delay 20.0;
      SEMO.SET;
      TX_B.MSG (M);
      SEMO.RESET;
      STATE := WAIT;
  or
    when STATE = WAIT =>
      delay 3.0;
      exit;
  end select;
end loop;
end TX_A;

```

This program leads to deadlock; the programmers did not understand that the semaphores, while they synchronize the processes' calls to each other, do not avoid simultaneous attempts to make an entry call if both tasks reach a 'SEMO.SET' statement at the same time. In this case, one task will grab the semaphore and try to call the other one, but that task will never be ready to accept the call since it is waiting at the 'SEMO.SET' statement, resulting in deadlock.

The study of these problems is interesting for several reasons. First, it indicates the caution to be taken when mapping a formal specification into an implementation; it's never as straightforward as it looks. Second, the diversity of solutions was surprising; in fact, the programmers' solutions covered the spectrum of possibilities suggested in various references for implementing asynchronous communication in Ada [BLW 87]. While the programmers found the tasking construct easy to grasp, they also found it was very easy to make subtle and hard-to-fix errors. The problem areas suggest certain points that should be emphasized in training for concurrent programming in Ada: the role of tasks in program design, the rendezvous as a synchronous communication model, methods for achieving asynchronous communication, the polling bias of the task construct, an explanation of the run-time implementation of tasks, models for understanding the dynamic concurrency in a system, and methods to avoid and detect deadlock.

4. Conclusion

This paper has presented lessons learned in teaching the Ada tasking model to experienced programmers in a distributed software experiment. Concurrent programming in the context of the Ada task proved to be a challenging exercise for the programmers, although all were enthusiastic about the Ada language when the experiment finished. We discovered many pitfalls for the unwary programmer in mapping formal design specifications into Ada implementations. A hands-on practice exercise in implementing a concurrent program from formal specifications was particularly helpful in pinpointing problem areas, although a dynamic model (such as petri nets) would have been beneficial in explaining the operational semantics of the rendezvous. The diverse approaches in implementing the practice exercise was surprising, and some interesting programming errors were discovered. It is clear that clever well-trained programmers will be important in extending software engineering into the distributed and concurrent programming domain.

Bibliography

- [AVK 84] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol 17 No 8, August 1984.
- [BLW 87] A. Burns, A. Lister, and A. Wellings, *Lecture Notes in Computer Science: A Review of Ada Tasking*, Springer-Verlag, New York 1987.
- [CCI 88] CCITT, "SDL, Specification and Description Language," (*Blue Book*) Z.100, *International Consultative Committee for Telephony and Telegraphy*, Geneva, March 1988.
- [CDG 86] R. Castanet, A. Dupeux, and P. Guitton, "Ada - A Well Suited Language for the Specification and Implementation of Protocols," *IFIP Workshop on Protocol Specification, Testing and Verification*, edited by M. Diaz, Elsevier Science Publishers (North-Holland), 1986.
- [ISO 84] ISO 7498, "Basic Reference Model for Open Systems Interconnection, International Standard, ISO 7498, Geneva 1984, also CCITT Recommendation X.200.
- [ISO 86] ISO/TC 97/SC 21, "OSI Conformance Testing Methodology and Framework," ISO DP 9646, edited by D. Rayner, Egham, September 1986.
- [ISO 87] ISO/DIS 9074, "Estelle: a Formal Description Technique based on an Extended State Transition Model," ISO DIS 9074, 1987.
- [ISO 87a] ISO/DIS 8807, "Information Processing Systems - OSI - LOTOS - A Formal Description Technique for the Temporal Ordering of Observational Behavior," ISO Draft International Standard 8807, October 1987.
- [ISO 88] ISO, "Guidelines for the Application of Estelle, LOTOS and SDL," Project ISO/TC 97 / SC 21, edited by K. Turner, Stirling, January 1988.
- [KEL 88] J.P.J. Kelly, D. E. Eckhardt, A. Caglayan, J. C. Knight, D. F. McAllister, M. A. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results," *18th Annual International Symposium on Fault-Tolerant Computing*, June 1988.
- [KM 89a] J.P.J. Kelly and S.C. Murphy, "Achieving Dependability Throughout the Development Process: A Distributed Software Experiment," *Submitted for publication*, 1989.
- [KM 89b] J.P.J. Kelly and S.C. Murphy, "Applying Design Diversity During System Development: An Experiment using Back-to-Back Testing," *Submitted for publication*, 1989.

Ada: Helping Executives Understand the Issues

David A. Umphress
Department of Mathematics and Computer Science
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

April 26, 1989

1 Introduction

In today's Air Force, we find an interesting dichotomy: junior officers at the grass roots of software development and senior officers making major software-related issues are Ada advocates; middle managers have been reluctant to accept Ada. The reasons provide an equally interesting view into corporate sociology. Junior officers are coming increasingly from computer science backgrounds and can appreciate Ada's software engineering features. Senior decision makers, even though they don't have a computer background, see standardization and interoperability as beneficial to large scale weapons systems development. Middle managers, on the other hand, typically grew up in the "slide rule" era, aren't always totally comfortable with computers, and often are reluctant to embrace new technology.

This paper describes experiences in helping Air Force middle managers understand the issues raised by Ada, both from a managerial and technical standpoint. The paper is divided into three major parts. The first part describes the material taught to a group of middle managers. The second part outlines the feedback from the group. Lessons learned in teaching the class are included in the third part.

2 CCSES

The Communications-Computer Systems Executive Seminar (CCSES) is a course held eight times a year at Maxwell AFB, Alabama, to acquaint upper-level middle managers with communications and computer issues in the military. Students in the class are Air Force Colonels and Lieutenant Colonels, or civilian equivalents, who are qualifying as directors of information systems. The course is two weeks long and features guest lecturers from within the Air Force and

Department of Defense. Of relevance here is the one hour session entitled "Ada: issues for executives."

The purpose of the lecture is heighten the awareness of students to the managerial issues of software development, in general, and Ada, specifically. Plainly, little real instruction can be accomplished in one hour. However, the time is used to point out the complexity of software and how that complexity is affected by using Ada.

Before the class, the students are furnished a copy of [Sammet 1986], DoD Directives 3405.2 and 3405.1, and a copy of the briefing slides used during the lecture. The lecture itself is divided into four major topics: background, features, issues, and technology assessment. The charts for the lecture are included at the end of this paper. Listed below is a brief description of each chart.

2.1 Background.

In the background portion, the students are given a working definition of software engineering. They learn that software development is a complexity management problem and efforts must be made to effectively minimize the impacts of human inability to deal with intensely intricate problems. They hear that embedded systems consume a majority percentage of software efforts in the DoD. They are then reminded that Ada was developed specifically to deal with embedded software. The rationale behind this discussion is to point out that Ada did not simply evolve, but was created consciously from a set of requirements.

- **Chart 1.** The objective of this slide is to show that there is a team devoted to software engineering with Ada and that the team is part of a recognized DoD agency. Many students are surprised to know of the organizational structure behind Ada. Many feel Ada "died off" long ago.
- **Chart 3.** This is a gentle introduction to the idea that software is deceptively difficult to develop. The term *software crisis* is, by now, a buzzword to all Air Force personnel involved with computers. This slide reminds the students that software development is an immature field and is fraught with problems. The last phrase, "[the electronic industry] has created the problem of using its product," always gets a few nods from even the most inexperienced people in the audience
- **Chart 4.** This slide was taken, in part, from the introductory Ada course taught at Keesler Air Force Base. The point made here is that the items under the "why" bullet are really symptoms. The real problem is that of managing complexity, both technically and managerially.
- **Chart 5.** Understandably, the term *software engineering* means different things to different people in the class. When this lecture was first offered, the official IEEE definition of *software engineering* [see IEEE 1987] was

used. The definition was vilified by the students as "techno-speak" and was subsequently dropped. This chart gives a definition with which the audience can identify. It points out that software engineering is a process to solve problems, not just a means to produce software.

- **Chart 6.** This is a "motherhood and apple pie" slide intended to point out that Ada was developed to fit a need in the DoD.
- **Chart 7.** As little time as possible is spent on this chart. While it presents some interesting milestones in history, it is generally well-known information. The points brought out by the slide are first, that Ada was developed in response to a valid need; second, that some attempt was made by the government to mandate the use of Ada in two DoD directives; and third, that compilers undergo a validation process.

2.2 Features.

In the features portion of the lecture, the students learn that Ada has characteristics that stretch beyond codification. The idea here is to emphasize that Ada embraces a systemic development philosophy of not only code, but software environments and methodologies as well.

- **Chart 8.** This slide points out that Ada has many of the same features as more "traditional" languages such as FORTRAN and Pascal. It also serves to illustrate that Ada isn't an entirely foreign programming language.
- **Chart 9.** The unique aspects of Ada are depicted here. Interestingly, this is as technical as the lecture gets and yet many of the students complain about the "high technical content" of the period.
- **Chart 10.** The objective of this slide is to point out that the potential, albeit currently unrealized, for software life cycle support is what should key managers to taking a closer look at Ada.

2.3 Issues.

The issues part of the hour is mainstay of the hour. It is here that the technical features of Ada as they relate to managerial control are discussed. Issues include design support, effects of code reusability, software performance criteria, hardware support, training, and configuration management.

- **Chart 11.** At this point in the lecture, the audience has warmed up and given a few Ada "horror stories." It is here that it becomes necessary to point out that problems exist, but they aren't always technical problems.

- **Chart 12.** The goal of this slide is to help the students understand that “armchair” Ada experts are easy to find. However, if they want the “*real thing*”, they need to allow their programmers (and managers) to experience Ada — in a structured software engineering education environment, for best results.
- **Chart 13.** This chart is used to illustrate that Ada has an impact on the managerial aspects of software development. The difficult point to get across is that managers don’t need to become intimately involved in the syntax of the language; they need to know the philosophy of the language and how to *manage* technical aspects.
- **Chart 14.** This chart generally receives the most attention. The objective is to discuss compiler reliability and tool availability. It is pointed out that compilers have matured greatly in the past years. Bugs, while still present, are moderately rare — much more so than with FORTRAN at the same point in its development. It is also necessary to point out that code produced by Ada is larger and potentially slower than that produced by other language compilers. However, on the other hand, a lot of run-time error checks are made that may not be available with other languages. Finally, tools are relatively scarce and expensive.
- **Chart 15.** Since the previous slide focused on the negative aspects of Ada, this chart attempts to address the potential long-term payoff features of the language. This information comes from [Foreman and Goodenough 1988]. Again, the emphasis is on the idea that there are no “quick fixes” to the software problems; long-term solutions must be examined.

2.4 Technology Assessment.

The last part of the period is devoted to a frank assessment of the current state of Ada.

- **Chart 16.** This chart illustrates some of the good and bad aspects of Ada with which managers should be aware.
- **Chart 17.** The bullets on this slide briefly summarize the briefing.
- **Chart 18.** This is a “let’s part on good terms” slide that pokes fun at the skeptics while emphasizing the mandate that Ada be used.

3 Feedback

The Ada lecture itself occupies little more than one percent of the course. However, it frequently turns out to be the most controversial and memorable topic

presented during the two weeks. Feedback in the class and on end-of-course critiques reveals a fascinating insight into what effort will be required to utilize fully Ada in the workplace:

- Few middle managers have had actual experience with Ada. They feel that Ada is just another programming language.
- They come to class with a preconceived notion that Ada is being foisted on them — and they resent the language because of it.
- They believe literature on Ada only when it attests to a purported project failure. They have “heard,” mostly through informal channels, that Ada is inherently slow, cumbersome, and only useful for embedded weapons systems.
- They often associate project failure with the Ada language itself, not possible managerial faults.
- Many feel they must know the syntax of the language in order to oversee projects effectively.
- Many do not believe that software engineering affects software development in a positive manner.

Whether or not these impressions are an accurate assessment of reality is not important. They represent concerns that middle managers have in working with Ada.

4 Lessons Learned

The lecture was originally based on [Murtagh 1988]. However, in over a year of presentation, the format and content of the material has changed dramatically. Lessons learned in teaching the class have been hard-earned:

- Middle managers do not want nor necessarily need to learn about specific language features. At best, general features of the language that support widely accepted software engineering principles should be presented.
- Ada should not be “sold.” Managers are, more often than not, astute students. They want objective facts. They expect to see not only good aspects of Ada but also the negative points as well.
- The “Ada story,” though often repetitious to many, is necessary to building the proper framework from which positive gains of the language are attained. Managers understand requirements; they can appreciate a language that was developed from predefined requirements.

- Managerial issues should be taught by someone who is a manager. Middle managers often resent being told about the managerial issues of Ada from technical people. They are afraid of "techno-speak."

5 Conclusion

The CCSES class has provided a rich insight into a realm of Ada that is not easily addressed by reference manuals and other technical information. However, the managerial issues of Ada are, and will continue to be, the front on which major battles in the use of Ada are won or lost. Much work must be done to address the concerns of the middle manager.

6 References

- Foreman, John, and John Goodenough. 1987. *Ada Adoption Handbook: A Program Manager's Guide*. CMU/SEI-87-TR-9. Software Engineering Institute, Pittsburgh, PA.
- IEEE. 1987. *Software Engineering Standards*. Institute of Electrical and Electronics Engineers, Inc., NY.
- Murtagh, Jeanne L. 1988. Presentation at Third Annual ASEET Symposium (Denver, CO, June 1988).
- Sammet, Jean E. 1986. Why Ada is not just another programming language. *Communications of the ACM* 29, 8, 723-731.

**Ada:
Issues for Executives**

**David A. Umphress, Ph.D.
Major USAF**

**Ada Software Engineering Education and Training Team
Ada Joint Program Office**

1

**Ada: Issues for Executives
Overview**

- Why Ada?
- What Is Ada?
- Concerns
- Technology Assessment

ASSET Team

**Ada: Issues for Executives
Software Crisis ... the problem**

- As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them -- it has created the problem of using its product
E.W. Dijkstra

ASSET Team

2

**Ada: Issues for Executives
Consequences**

- Software characteristics
 - Expensive
 - Unreliable
 - Late
 - Not maintainable
 - Not transportable
- Why?
 - Too many languages?
 - Changing technology?
 - Not enough trained people?

INABILITY TO MANAGE COMPLEX PROBLEMS

ASSET Team

Ada: Issues for Executives
Software Crisis ... the solution path

- Software Engineering is ... applying common sense and engineering discipline to solve a problem in such a way that the solution doesn't violate common sense.
- ... the by-product -- software -- should support computing which is
 - Realistic
 - Economical
 - Reliable
 - Understandable

ASSET Team

5

Ada: Issues for Executives
Software Crisis ... Ada, a solution?

- DoD needed a tool that
 - Addressed development of software for embedded weapons systems
 - Enforced application of good software engineering principles
 - Supported entire software life cycle

The Result: Ada

ASSET Team

6

Ada: Issues for Executives
A Brief History

1975	HOLWG
1977	4 Design Teams
1978	Steelman 2 Design Teams
1979	Honeywell CII Bull Ada Ada Joint Program Office
1983	MIL-STD-1815A First Translator
1987	DoDD 3405.2 & 3405.1
1989	>100 Validated Compilers

ASSET

7

Ada: Issues for Executives
Features of Existing Languages

- FORTRAN
 - Scientific notation
- Pascal
 - Structure control constructs (e.g., if-then-else, case, while)
 - Strong data types
 - Dynamic memory allocation

ASSET Team

8

Ada: Issues for Executives Unique Features of Ada

- Packages
 - Procedural/data modularity
- Exceptions
 - Traps for run-time errors
- Generics
 - Templates for reusable code
- Tasking
 - Parallel computing

ASSET Teams

9

10

ASSET Teams

Ada: Issues for Executives And, Most Importantly ...

- Software development support
 - Development environments
 - Development methods

Bottom Line: Ada was designed to address a broad range of issues in the software life-cycle

Ada: Issues for Executives Concerns

- Characteristics
 - It's new
 - It's big
 - It's complex
- Major Areas Impacted
 - Training
 - Software support

ASSET Teams

11

12

Ada: Issues for Executives The Programmer -- Ada Training

- What?
 - Full understanding of the language
 - Actual experience with the language
- Why?
 - DoD 3405.2 and 3405.1
- How?
 - Ada compilers at work
 - Release for training programs
 - Experiment with systems at work

ASSET Teams

**Ada: Issues for Executives
The Manager -- Ada Training**

- What?
 - Basic idea of the Ada philosophy
 - Understand issues and problems
- Why?
 - Using Ada affects schedules
 - Using Ada requires resources
 - Using Ada requires education

ASSET Doc#

13

**Ada: Issues for Executives
Then Why Use Ada???**

- Code portability
- People portability
- Maintainability
- Reliability
- Common basis for tools and methods
- Management visibility
- Productivity

ASSET Doc#

15

**Ada: Issues for Executives
Software Support**

- Compiler issues
 - Ease of use
 - Speed of code produced
 - Bugs
- Tool issues
 - Integration
 - Availability
 - Effectiveness

ASSET Doc#

14

**Ada: Issues for Executives
Technology Assessment**

- da good
 - Excellent software engineering support
 - Addresses more than just "coding"
 - Compilers maturing
- da bad
 - Weak government/academic support
 - Trained people
- da ugly
 - Technical gimmickry must be supported by managerial strength

ASSET Doc#

16

Ada: Issues for Executives
The Bottom Line

- Effective software engineering tool
- Technical and managerial planning necessary to use Ada
- Technology not fully developed, but progressing rapidly

ASSET Team

17

Ada: Issues for Executives
The Future Isn't What It Used To Be

- Everything that can be invented has been invented.
Charles Duell, 1899
US Patent Office
- Who the hell wants to hear actors talk?
Harry Warner c1927
Warner Bros. Inc.
- Sensible and responsible women don't want to vote.
Grover Cleveland 1906
US President
- We don't need to worry about Ada. Mark my words, it'll go away.
Your name?
CCSES Attendee

ASSET

18

The Pedagogy and Pragmatics of Teaching Ada as a Software Engineering Tool

LCdr Melinda L. MORAN
NARDAC Washington
Washington, D.C. 20374
morani@ajpo.sei.cmu.edu
(202)475-7681

MAJ Charles B. ENGLE, Jr.
Software Engineering Institute
Pittsburgh, PA
engle@sei.cmu.edu
(412) 268-6525

Introduction

Among the decisions currently facing many computer science departments is the choice of a primary language of illustration and implementation. This decision is being couched in the context of a much larger environment of change being proposed by a joint ACM/IEEE-CS Curriculum Task Force. The changes being proposed by this task force are pervasive and, at this point, still subject to review and revision. One point which is clear, however, is that the thrust of the proposed changes is to shift the focus away from "computer science as programming" to "computer science as a much broader discipline." In the same vein, although many elements of this paper focus on the pedagogy and pragmatics of teaching Ada, the language, by far the more important focus is on teaching Ada, the software engineering tool.

A Tool for Programming-in-the-Large

One of the most important pedagogical points to be made about teaching Ada as a tool for software engineering is that it is imperative that the language be introduced in the context of programming-in-the-large with emphasis on the semantic constructs of the language, NOT in the context of programming-in-the-small with emphasis on the syntactic constructs of the language. A "need" for the language must be developed in the student before any true appreciation of the language can be gained. (This need would be called by psychologists a "cognitive dissonance.") Without a perceived need the student is less

motivated to learn and extend what (s)he has been taught to the solution of new problems.

Among computer science students, even in a CS1 class, it is increasingly impossible to find anyone without some prior exposure to a computer and/or some programming language. Almost all students enter the curriculum with functional mastery of at least one computer language, normally BASIC and/or Pascal. The instructor who attempts to introduce the Ada language to students using the traditional "toy" programs used in prior years in introducing BASIC and Pascal is doomed to failure. (S)he is immediately met with resistance from students asking "Why should I worry about learning how to do this in Ada when I can already program this in BASIC or Pascal?" There is validity in this question. If the semantics of BASIC or Pascal are simply translated into the syntax of Ada students have gained nothing. Indeed, "...a software crisis has not been caused by a lack of syntactic knowledge. The problem lies in the scope of our SEMANTIC knowledge and our ability to apply it. The goal of software engineering is to expand knowledge of the semantics of software development." [PRE87]

How then do we create a "need" for Ada within the student? The pedagogy is simple and will be elaborated on first. The pragmatics, unfortunately, are weighty and so we defer a discussion of them until later.

One of the major semantic differences Ada introduces is a significant emphasis on modularity. The package construct facilitates the construction of software systems composed of many modules. The package is a masterful tool for embodying the use of layers of abstraction and information hiding within a system. It illustrates clearly to students the logical separation and encapsulation of related resources.

This massive modularity (a very real part of most actual systems) is usually entirely foreign to most introductory students. Not having a program complete in one module makes most students highly uncomfortable. (Good! Cognitive dissonance!) The first project presented

to introduce students to Ada (and to software engineering) should be a large-scoped one requiring MANY modules and a great deal of coding. The instructor should guide students through the design phase initially leaving them with the impression that they will be actually coding the implementation of each module. Panic will ensue almost immediately. Students will be unable to cope with the level of complexity and that will be immediately apparent to the vast majority of them. Before outright rebellion occurs, however, the instructor should step in and provide students access to functional versions (but not necessarily source code) of all required modules except the main driver module. The main driver module should require minimal coding; it is likely to be simply a routine which utilizes the resources encapsulated in the provided modules. Students are now back at the level of "coding in the small" but have gained an appreciation of building systems "in the large" because they can see the larger picture of how their module and all the others comprise the larger system. Further, they realize the lack of constructs in Pascal and BASIC to modularize easily on such a large scale. They have an appreciation for one of the major SEMANTIC extensions in Ada and are not locked in the emotional resistance engendered in simply teaching Ada as a syntactic transition from Pascal and/or BASIC.

The Taboo of Teaching by Analogy

As an example of the problems which can be encountered when Ada is not taught in this manner, consider the experience of one Ada instructor with whom the authors are familiar. This instructor was new to Ada when Ada was also new and decided to teach Ada as just another programming language. He employed the technique of teaching by analogy using the programming language Pascal as the basis for his analogies. Starting at the lexical level, he had completed his instruction in Ada's typing mechanisms and control structures and had just about finished a description of subprograms when the student's first programming assignment was handed out. Given the limits on what the students had been taught, the assignment was designed to be an Ada (main) procedure which made use of several embedded procedures. The structure of this style of programming, naturally, resembles a Pascal program. Not

surprisingly, the students had little or no difficulty in completing this first assignment properly.

The instructor then began a detailed discussion of the Ada package and why it was both important and necessary for programming-in-the-large. After several class discussions on packages and their uses, a second assignment was given to the students with the stipulation that a package was required for the solution. One student turned into the instructor a complete assignment in which the package specification listed all of the subprograms needed for the solution to the programming problem. In the body of the package all of the subprograms were correctly implemented. However, the student had developed his entire algorithm for the problem solution between the begin-end block of the package body, i.e., in the portion of the package body which is designed for initialization. The statements listed in this part included input/output calls for interactive dialogs with the program user, declare blocks with generic instantiations, and many other features which are not traditionally thought of as being appropriate for the "initialization" portion of the package body. When the student had completed compiling this package, he had tried to link it. Of course, it could not be linked and the compiler output some cryptic message to the effect that the package was a "passive" entity and could not be linked in this manner. After consultation with another student, the student that wrote this package discovered that he needed to have a subprogram as the "driver" and that he needed to "with" the package he wrote in this subprogram. He therefore wrote a procedure with a body that had a single statement, namely **null**. When he compiled and linked this procedure and then ran it, his solution to the programming problem worked perfectly. It seems that since his null procedure "with"ed the package, the package was elaborated at run time. Elaboration includes the execution of any statements in the "initialization" portion of the package body. Since the student had implemented his entire algorithm at that point, he naturally got the "effect" of program execution, although in reality his program never executed, it merely elaborated! This led to several lectures devoted to explaining what had happened and why a programmer would not want to program in this manner. Needless to say, many students never did understand what the difference was, nor what had happened in this case.

The cause of all of this confusion was the instructor's approach to teaching the language. The student knew Pascal and by analogy tried to convert Ada into Pascal. The result was "Adacal"; the student was thinking in Pascal but coding in Ada. The moral: **Never start teaching Ada "bottom-up."** Ada embodies an entirely new approach to programming and must be taught in an entirely new way to break old habits early. The need for the package and its role in modularizing large programming solutions **MUST** be the first thing taught to students. Lower level features can then be added in a coordinated manner as discussed in the rest of this paper.

Creating Courseware

The pragmatics of introducing Ada in this "top-down" programming-in-the-large manner are weighty. The instructor must design a large project and then make many of the requisite project modules available to the students. Obviously the instructor must somehow create the modules which are given to the students. The project must be large enough that a student could not conceivably create all the required modules in the allotted time. This demands a great many modules, with the actual number dependent upon the abilities of the student population. A good deal of time must be invested by the instructor in creating these modules. This time should not be trivialized. To create well-styled, well-documented, robust modules takes a great deal of effort. Certainly the increasing availability of commercial libraries of generic units, the growing number of bulletin board sources, etc. helps in reducing this creation time somewhat, but the instructor must still invest time integrating and testing acquired units.

In an effort to assist instructors in this endeavor, the Software Engineering Institute (SEI), a federally funded research and development center at Carnegie Mellon University, is attempting to provide artifacts to be used for instruction. These artifacts are currently all written in Ada and are available for the cost of reproduction (generally less than \$10). They are between 10,000 and 20,000 lines of code, depending on how a line of code is counted. Each artifact is self-contained and consists of the

source code for the artifact, all of the documentation for the design and development of the artifact, a test suite, and some tools to make the instructor's use of the artifact less difficult, such as compilation order listings. Examples of the documentation which come with each artifact are the user's manual, the test plan, the configuration management plan, the requirements document, preliminary and detailed design documents, quality assurance plan, coding standards, and documentation standards.

This effort by the SEI Education Program was undertaken specifically to address the need by instructors for large project courseware. The provision of this courseware allows students to concentrate on those aspects of learning that the instructor is trying to impart and frees them from the intellectually less stimulating busy-work of implementing modules needed to execute the modules of interest for a specific lesson. It prevents students from becoming immersed in minutiae and focuses their attention on the important instructional points. Students can concentrate on "building" the program and not on erecting the scaffolding needed to build the program.

Library Managers

Having courseware available, however, is not enough. There must be some mechanism for affording students access to the functional implementations of the modules. If a mainframe environment is being used, most library managers provide some mechanism for students to simply enter a pointer to the appropriate object modules rather than requiring each to maintain their own copy of the module. If a microcomputer environment is being used, some mechanism must be developed to allow students to copy the provided modules into their own storage area.

Textbooks

Another pragmatic consideration in introducing students to Ada in the aforementioned manner is that of finding a textbook which supports this pedagogy. First, finding a textbook which focuses on the constructs important in CSI and secondarily uses Ada as a tool to introduce and

illustrate those constructs is impossible. One does not yet exist. Currently there are CSl books and there are Ada language books and the selection of one approach or the other is an exclusive-or situation. Let us make the optimistic assumption that student economics are such that this problem is solved by having students purchase two books, an idealistic assumption certainly. Even finding an Ada language book which begins at the level of packages and introduces the syntax secondarily by having students program in the small to complete small parts of much larger systems is difficult. Most Ada language books begin at the syntactic level focusing on the lower end of the language. Many even introduce syntactic constructs by juxtaposition with their Pascal counterparts. These books, in these authors' opinions, teach students to think in Pascal semantics and write Ada syntax (Adacal?), a worthless accomplishment.

The lack of a supportive textbook, again, is not an insurmountable problem, but certainly one meriting attention. Many textbooks can be made suitable by conscious reordering of the material by the instructor in assignments. In resequencing material, however, careful attention must be paid to contextual dependencies inherent to the author's original ordering. Beginning students, unfortunately, tend to be easily panicked by encountering unfamiliar constructs OTHER than the one they are supposed to be learning in a specific lesson and are easily distracted from the primary focus of the lesson.

Another problem in finding an appropriate textbook is, again, lodged in finding a book which is not simply a conversion effort from a previous language. Beyond teaching Ada syntax coupled with an older language's semantics, most of these books also centre almost entirely around a functional decomposition approach to design. Creating software systems written in Ada is certainly not incompatible with this method of design, but the use of object-oriented design is decidedly more effective in creating modular well-designed systems.

Finally, on a more minor scale problem, finding a textbook which introduces the richness of Ada's user-defined types and subtypes and then continues to use these types in examples throughout the book is difficult. A number of books exist which introduce these types early and espouse the

virtues of utilizing them to create good abstraction and model reality. Almost all of these books immediately revert to the use of standard predefined types in all subsequent examples. Another case of the authors' conversionism showing through?!

Modularity and the Demand for a Decent Environment

Another pedagogical point deserving careful attention when introducing Ada also stems from the significant emphasis on modularity facilitated by Ada (and good software engineering practices.) Students must be introduced to the concept of a library and to the concept of compilation dependencies. Often, depending upon their background, students must also be introduced to the idea that compilation, linkage, and execution are distinct processes. Many students coming from BASIC and/or Pascal backgrounds are oblivious to this idea; they are familiar only with choosing the RUN option and having compilation, linkage, and execution all occur instantly. This classic use of "information hiding" by many environments interferes with a student's comprehension of compilation dependencies unless the information hiding is dispelled.

There are a number of points worth making about introducing students to the concept of libraries and compilation dependencies. The first is that an easily navigable library manager and a good environment which displays compilation dependencies can be very helpful to students learning this construct; the environment may even provide an automatic recompilation facility which takes these dependencies into account. The second is that the environment should provide an editor with a multiple windowing capability to facilitate modular program development. The student should be able to easily call up and view other program units while concurrently developing a dependent unit. Having to endlessly flip flop back and forth between files in the editor is tremendously frustrating to beginning students and introduces a hostility which is misdirected at the language itself. From a pedagogical view these two points are important. From a pragmatic view they are not so easily realized. Validated compilers are now plentiful and a number of affordable student versions exist for microcomputers as well. Well developed environments which encapsulate a "student-friendly,

student-fast" editor and compiler are much sparser and, those that do exist, are not yet "student-affordable." This situation is changing daily, however, and should be constantly reevaluated in deciding to shift to Ada as a primary language.

What is needed is an environment which is integrated. It should be possible for the student to create a module in the editor, send it to the compiler, and, upon an unsuccessful compilation, be placed immediately back in the editor positioned at the point of the error. Similarly, upon successful compilation, the student should be able to link and execute the module all within some enclosing framework. In addition, this framework should be able to construct, from source code, the compilation dependencies. The student should be able to play "what if" games in seeing the effects of re-compilation on the rest of the modules in the set needed for program execution. An automatic recompilation capability should also be included. All of these features, and more, are currently available on most mainframe implementations of Ada. To date, some microcomputer implementations of Ada have started to make some, even most, of these capabilities available with their compilation systems. Instructors should seek out these implementations and give them preference in the selection of a compiler for the student use.

The Wisdom in Avoiding Use of "Use" Clauses

A final pedagogical point also needs to be made on the introduction of students to massively modular Ada programs beyond the demands introduced for a decent environment. This pedagogical consideration is the timing involved in introducing students to the use of the "use" clause. Introducing students early-on to the use of the "use" clause can be as crippling a mistake as choosing a poor environment. Requiring students to use expanded dot notation in identifying the source package of each resource used in a program is highly recommended for at least the first half of a CSI course. Students are unfamiliar with the massive modularity introduced in Ada programs. The ambiguity added by allowing them to utilize the "use" clause early-on in their experience with Ada compounds their problems in clarifying the interdependencies of units. When

compilation errors occur it compounds their difficulties in identifying where to look to resolve these errors.

Generics and I/O

Moving from the focus on modularity, there is another pedagogical consideration which must be addressed early in teaching Ada. That is the issue of teaching students how to do I/O in Ada.

There are a number of schools of thought on the most effective (and least painful) way of introducing students to how to do I/O in Ada. One school supports the creation of a Basic_IO package by the instructor which is then provided to the students at the beginning of the course. In Basic_IO are all the instantiations necessary to provide Get's and Put's for all predefined data types. The students then simply "with" this package and "automagically" have the ability to do I/O. A second school supports simply mechanically instructing students on how to write the statements necessary to instantiate and "use" an I/O package. In this method, total information hiding is employed as to what generics is and what instantiation does.

Having been badly burned as a follower of the first school of thought, one of these authors is heartily against providing students with a Basic_IO package. Such a package loses its utility very quickly in the curriculum, as soon as user-defined types are introduced. (And these should be introduced, for purposes of good abstraction, VERY early.) Further, giving students a Basic_IO package is providing a crutch which is almost impossible to wrest away later in the curriculum. Experience has shown this author that the second school, modified with a minimal and concrete analogy of what generics is and what instantiation does, is infinitely preferable in teaching students how to do I/O in Ada.

Economic and Inertial Forces in Teaching Ada

Two final points merit mention in deciding whether to select Ada as primary language of illustration and implementation in an undergraduate

computer science curriculum. Both are pragmatic considerations. One is a very transitory consideration which must be constantly re-evaluated. The other is a very weighty consideration rooted in the nature of human beings.

The economic demands placed on students in deciding to transition a curriculum to Ada should not be ignored. Ada compilers for microcomputers currently require a hard disk and 640K to function effectively. (Some will indeed function without a hard disk but the disk swapping involved to accomodate this is unacceptable.) Hard disks and additional memory are non-trivial investments for students. Unless sufficient capacity and easy access exists to support students in the use of a mainframe, these costs must be considered. These costs, however, are constantly decreasing and should be continuously re-evaluated.

The institutional inertia exhibited by many faculty members in changing the status-quo must also be considered in determining whether to transition a curriculum to Ada. This resistance to change and learning something new is very real! It has roots in the time vested by established instructors in existing lesson plans and courseware. It is a major impediment to overcome! It must be carefully considered in determining whether a successful transition to Ada can be made and/or will be accepted by a computer science faculty.

Conclusions

A myriad of factors, both pedagogical and pragmatic, must be considered in transitioning an undergraduate computer science curriculum to Ada as the language of illustration and implementation. There are many positives to be gained on the side of student learning. Students learn the benefits of packaging in creating systems which embody layers of abstraction and exhibit modularity which makes them very modifiable and maintainable. They learn the benefits of user-defined types and subtypes in modeling reality and creating systems which are therefore more understandable. They learn the benefits of exception handlers in creating fault-tolerant systems. They learn to function in an environment of programming-in-the-large, one such as they will undoubtedly encounter upon graduation into the "real" world of software development. These

benefits, in these authors' opinions far outweigh any negatives introduced on the pragmatic side.

Further, it is these authors' opinion that adequate awareness and preparation on the part of most computer science instructors can successfully avoid most, if not all, of the impediments recounted in this paper. The moral of this paper might indeed be summed up in the phrase: "*Learning Ada is easy; teaching Ada effectively is very demanding.*"

References

- [PRE87] Pressman, Roger S., *Software Engineering: A Practitioners Approach*, 2ed., McGraw-Hill, 1987.

Incorporating Ada Into a Traditional Software Engineering Course

Albert L. Crawford

Department of Computer Science,
Southern Illinois University, Carbondale, Illinois 62901

The Course

Student background

The students in CS435, Software Design and Implementation, at SIU are seniors or first year graduate students. They all know Pascal and several other languages. Among the most common languages that the students know are FORTRAN, COBOL, C, LISP, and Modula-2. The students have a good grasp of basic programming techniques for programming "in the small." They have been well versed in structured programming, top-down design, basic file handling and documentation of program code.

Many of the students have had some work experience involving program development through summer jobs or part-time jobs. Most of these jobs, however, were for small organizations whose programming needs were very limited. These jobs frequently involve programming of microcomputers using Lotus 1-2-3, Dbase III, or other similar software tools. In no case does the job experience of the student involve the development of large software systems.

The largest project that any student has worked with is usually under 5,000 source lines of code. Students have had little or no experience working within a programming team. The concept of 20 or more people working on the development of a software system for a few years is beyond what most of the students have ever considered.

Course objectives and structure

It is the primary purpose of this course to present to the students some of the concepts involved in the development of large software systems. The students develop of a

software system of about 10,000 source lines of code to achieve this goal. Each phase of the software lifecycle through implementation is emphasized. The lectures include the testing and maintenance phases, however these phases are not stressed within the project itself.

The first phase is to develop a program definition. A group of selected students write the program definition that the class uses during the semester. If enough graduate students are available, they form this definition team. Otherwise, the definition team consists of a group of undergraduates who have previously demonstrated superior skills. This definition team works with the instructor to develop a program definition suitable for the class. The rest of the students in class individually write a program definition of their own

In the second phase, each member of the definition team chairs a system analysis team. A random selection determines the membership of each analysis team. Each team performs the design and analysis of the problem assigned using the program definition developed during phase I. Each team performs the systems analysis using the object-oriented design paradigm. It is a requirement that the analysis teams define a system in such a way that independent implementation of each module according to specification guarantees that the integrated system would execute without error. The best of these system designs is chosen for use in phase III.

In the third phase, the winning system design team forms the management group for the implementation of their system. Added to this management team are two or three other class members. The rest of the class are then put onto implementation teams that implement individual modules of the system. Each team completes their implementations. The entire project is integrated and executed.

It is an objective of the course to study a single paradigm for the development of a large software system. The students have the experience of developing a system as a group

instead of as individuals. The management skills and the communications skills required for such group interaction are the most important concepts presented in this course.

Language Requirements

Necessary language features

The language used in a software engineering projects course should have several rather obvious qualities. The first, and most important, of these is the support that the language gives for modular programming. The language should explicitly support modular design. These modules should be separately compiled. The interfaces between the modules should be well defined and enforced by the language.

The second, and related, feature that is necessary for the course language is the support for information hiding and data abstraction. The use of object oriented programming requires the software engineer to define data abstractions on the objects of the problem.

These first two features are necessary to support the independent implementation of the modules. The completion of a relatively large software project to execution within a single semester requires the language to help the students with the modules independence. Otherwise the integration of the project would become impossible.

If the language is not a prerequisite for the course its presentation within the course must occur without interfering with the objectives of the course. The language should be similar to the languages already known by the students. Its teaching must compliment the primary course objectives. Students must feel that the language is a tool for the course but not an objective of the course.

Current possible languages

There are several languages that are satisfactory for the teaching of software engineering. Indeed, almost any language could be used with success if the instructor

wished. This author feels there are four languages now available that meet the course requirements. These are Modula-2, Turbo Pascal (versions 4 and above), C + +, and Ada.

Modula-2 is an excellent language for software engineering. It supports well defined modules and provides for data abstraction through its opaque types. In addition there are several low-priced Modula-2 compilers now available. There is also a large amount of instructional material available for Modula-2. Since Modula-2 is very similar to Pascal, the students learn it very quickly.

The latest versions of Turbo Pascal support a programming unit similar to an Ada package. This provides an excellent interface between modules. Turbo Pascal provides an excellent programming environment that is very user friendly. Turbo Pascal also provides graphics commands that allow the students to "jazz up" their programs. Since the students already know Pascal, and many of them know Turbo Pascal, teaching is no problem.

C + + uses classes for its modular development. These classes provide features such as inheritance that in some ways is superior to the other languages. However, only a limited amount of literature exists for C + +. Furthermore, many of the students do not know the language C so the teaching of C + + would be more difficult than the languages in the Pascal family.

Ada has come into its own in the past two years. There are now available low-cost Ada compilers. The primary drawback of Ada is the lack of material designed for the university classroom.

Other languages are feasible for a software engineering course. However, the other languages lack some of the most desired features. FORTRAN and COBOL have separate compilation but the interfaces are undefined within the language. The separate compilation units in C are adequate, but the compilers will not enforce the proper use of the interfaces. Standard Pascal does not support separate compilation.

Reasons for choosing Ada

Ada has several features that make it superior to the other languages for a software engineering class. The exception handling features of Ada are easy for the students to learn and to use. This allows them to insert error detection routines within their code without undue difficulty. This helps the students to develop modules that are reasonably reliable.

The private types of Ada are superior to the opaque types of Modula-2. This provides a much better tool for the introduction of data abstractions within the project. The use of Ada and the classroom presentations act together to reenforce the important software design concepts.

As a side benefit, the use of Ada increases the students' marketability when he/she graduates. Several students have reported that they were offered jobs because of their knowledge of Ada. While pedagogic considerations should be the primary reasons for the design of a university level course, it is always good if the pedagogic and economic goals produce the same result.

Ada Within the Classroom

Introduction to Ada

The introduction of Ada to the students occurs in two stages. The first stage is the presentation of a Pascal-like subset of Ada. This usually takes about two class periods. A very quick coverage of the syntax of Ada is the focal point of the lectures. In addition, the students learn the concept of a package from the client viewpoint. How to read a package specification, the WITH statement, the USE statement and the instantiation of a generic package are topics covered in this first stage. The package TEXT_IO is presented. The students write a simple program to become familiar with Ada and the Ada compiler (we now use the Janus Ada compiler.) This program uses the basic Ada statements as well as text file operations.

The second stage of classroom presentation occurs about three weeks after the first. The focus of this stage are the concepts of package development and exceptions. This requires only one class for Ada itself. However, the discussion of packages and exceptions occurs along with a discussion of data abstraction and information hiding. The total discussion usually requires about three class meetings. The students implement a simple data abstraction package, which includes package defined exceptions. This usually involves a stack or a queue package for which the students are given the package specification. The students write their own driver for the package for purposes of testing the package. The class teaching assistant also writes a driver for the package, which the students will not see until after the lab is due. The TA grades the lab by running both the student's driver and his own driver linked with the students implementation of the package.

The students will not be exposed to either tasking or generic packages within the class itself.

The use of Ada on the project

Phase I of the project, the problem definition, is programming language independent. The students do not depend on the style or constructs of any language to form the definition.

After the completion of the class project definition the instructor randomly divides the students into groups with about 5 members each. Each group does an initial system analysis using the object oriented design paradigm. The analysis team selects the objects of the project definition. The teams construct an Ada package specification for each object and its operations. Those objects for which multiple occurrences exist in the problem are implemented as abstract data types.

Each group is to complete a compilable set of package specifications along with a compilable main program. All entries in the specifications are defined by documentation so if they are implemented as defined the program system is guaranteed to execute

properly. In addition each team prepares bubble diagrams of the system. To assure themselves of the correctness of the specifications the students perform structured walk throughs of the system.

The group leader maintains a log of each meeting of the group. The instructor uses peer evaluations, the quality of the finished system design and the log for determining each individual students grade for the system analysis.

After the winning design has been chosen the students are then reorganized into groups. This time the entire class works on implementing the winning design. The leader of the winning design team is the overall project manager. The instructor and the project manager select an assistant manager from among other design team leaders. These two then oversee the individual groups during the implementation of the modules. They keep logs of their meetings as well as a paper trail of all specification changes that were necessary during implementation.

As the student version of Janus Ada will not all the linking of the entire project, we port the system to the Sequent Balance in the department. This machine uses the Verdix Ada compiler. Since the student version of Janus Ada supports only standard Ada packages, the porting takes place with absolutely no problems. Integration is usually reasonable quick. It normally takes three four hour sessions with the team leaders to accomplish the integration of the system. The first session finds the major problems of the system. The second session uncovers a few minor bugs. The last session results in an executable prototype of the project goal.

The Presentation of Tasking and Generic Packages

The formal presentation

The Ada concepts of generic units and of tasks are not presented in the classroom. These concepts are not needed for the student project and to spend class time on the concepts would intrude into the basic goals of the course. However, the students should

have the opportunity to learn these concepts. It is important that they are encouraged to develop a more complete understanding of Ada. Therefore, optional seminars on each subject present to the students the concepts of generics and tasking.

The first seminar is on tasking. This session includes the standard tasking concepts of tasks, task types, and rendezvous. Also presented are the rules of task declarations, task parents and task termination. The students get a simple lab requiring the declarations of several tasks. This lab may be done by the students for extra credit in the software engineering course.

The second seminar, on generics, covers the basic concepts of generics with the emphasis on generic packages. The seminar covers generic formal parameters detail. The students already know how to instantiate the generic packages such as INTEGER_IO and FLOAT_IO. The second seminar presents a more comprehensive coverage of this concept. Again, a simple lab is assigned which may be done for extra credit.

The student response

About 70% to 80% of the students attend the seminars. Most are very eager to learn about tasking. Since the concept of generic programming comes up in other courses, most are willing to attend the seminar on generics.

However, very few students actually do the extra credit labs. Only about 3 or 4 do one of the labs each semester. I have yet to have a student do both of the extra credit labs. The primary reason for this is probably that the extra credit labs are assigned toward the end of the semester. At this time the project is well under way. The semester project requires a large amount of each student's time.

Conclusions

One of the biggest problems with the course is the lack of a textbook that fully supplements the course. Ada books seldom contain the software concepts necessary. On the other hand, software engineering textbooks contain too much theory for this course.

Even the popular textbook "Software Engineering with Ada," by Grady Booch, has its shortcomings.

The suitability of the Ada programming language to software engineering is obvious. However, it is difficult to include Ada into a course without interfering with the primary purposes of the course. Many of the complexities of the language must be omitted. The complex typing system is presented in a simplified manner. As previously discussed, generics and tasking are not used within the class. Ada proponents notwithstanding, Ada is a more complex language than are most languages.

Even by presenting Ada in an incomplete form, however, the flavor of Ada still comes strongly through. If the instructor is careful, Ada can be used by the students without undue interference with the primary purposes of the course. The full benefit of Ada can be realized along with the objectives of this project oriented course. Ada is indeed the language of choice for software engineering.

MOTIVATION AND RETENTION ISSUES IN TEACHING ADA
or

How Will Students Learn Software Engineering When
Their Only Goal is Ada on their Resume?

ABSTRACT:

Every teacher must consider motivation and retention issues whatever the subject. This is especially true in teaching Ada because until now most Ada students have been programmers trained in other languages (FORTRAN, COBOL, Assembler) who have experienced success using ad hoc design methods. Teachers must therefore consider not only how to present new material but also how to overcome old habits. This paper will discuss how we addressed these issues at Atlantic Community College.

BACKGROUND:

Atlantic Community College is located in Mays Landing, New Jersey, near the FAA Experimental Center. In response to community demand we decided to explore teaching Ada as a credit course in the CIS curriculum. We immersed ourselves in Ada material before attending the "Software Engineering in Ada" workshop at the Air Force Academy in July of 1988.

After the workshop, when we discussed the idea of implementing a course in Ada, we realized that motivation and retention issues must be addressed in the planning stage. Most of our students are programmers who wish to upgrade their skills ("I want Ada on my resume" was a typical response when we asked the reason for taking the course). These students worked all day and then would come to class for three hours. We planned from the beginning how to motivate these students and retain them throughout the course while still insisting on valid Ada training.

Our initial course in Fall of 1988 had full enrollment of thirty students, twenty-seven of whom successfully completed the course. Ten of the students wished to immediately continue with Advanced Ada and so in Spring of 1989 we had another full section of Ada 1 (in fact we had an overload) and a section of Ada 2.

We have learned from these experiences what works and what does not.

CURRICULUM DESIGN:

We began by asking ourselves questions. We would like to explore in this presentation our answers. The following is a brief outline of our questions and responses.

1. What teaching style should we use?

ISSUE: We believed in teaching Ada using the concepts of software engineering. What teaching style would allow us to deal with theoretical considerations without putting our students to sleep?

RESPONSES: We decided that a team teaching style was most appropriate. We had experienced team teaching as students during the Air Force class and during an ASEET workshops and we felt that the interaction between teachers was one means of keeping student interest.

2. How should the course be structured?

ISSUE: How could we give students frequent feedback and secure feedback for ourselves?

RESPONSES: We give detailed handouts with behavioral objectives so students know exactly what is demanded of them. We also give weekly quizzes, each of which contains a programming part and a theory part.

3. How could we get students to do valid programs?

ISSUE: Most of our students programmed all day long. They were resistant to doing more for school, yet we knew from our own experience that unless they wrote programs they would not learn Ada.

RESPONSES: We had most success in incorporating simulations into our assignments. We had students translate Morse Code and play hands of bridge in Ada. These were not the type of programs they did at work -- they were "fun". We copy the best programs and distribute them to the class for discussion.

4. How can we stress software engineering concepts?

ISSUE: We want students to use the tools in Ada that support software engineering, not just "Code a FORTRAN program in Ada".

RESPONSES: Use the weekly quizzes to test concepts, and use the program assignments to test application. During the Fall of 1988 we gave five separate programming assignments, but during the Spring of 1989 we give five related assignments in which each part builds on the previous part. We also purchased the Ada repository and make use of the libraries.

5. How can the teachers serve as role models?

ISSUE: We do not want to preach what we don't practice.

RESPONSES: We do every assignment we expect from students. We have also set ourselves the task of working on a large assignment cooperatively to expand our software engineering skills.

ATLANTIC COMMUNITY COLLEGE
Course and Faculty Evaluation
WRITTEN SEGMENT

All written answers will be typewritten before being reviewed by the instructor at the end of the semester.

1. List the things you like most about this course.

Appropriate subject matter to job.

Good student participation.

Overhead audio visual use.

Availability of compiler (to purchase reasonably).

Instructors - prepared, good directions.

Weekly quizzes.

Multitude of examples.

Assigned programs.

Handouts.

Two instructors - good idea.

2. List the things you like least about this course.

Text - errors.

Only one computer available in lab.

Opportunity to correct and resubmit tests.

\Alice\fultime.evl

Language Experience
AAA I

Assembly (8.4%)

BAI (11.3%)

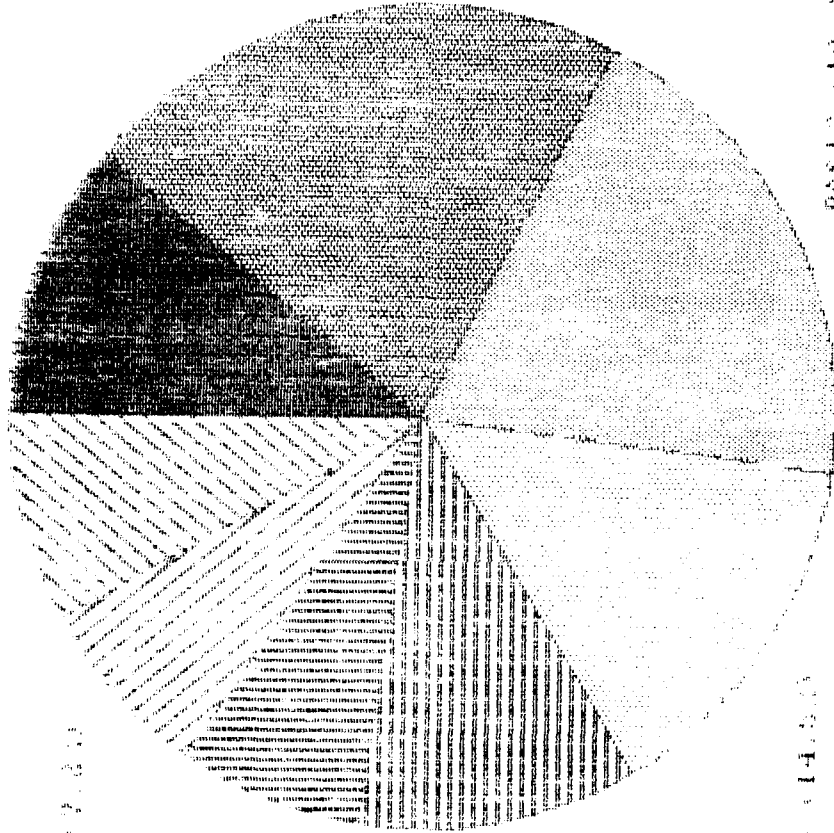
CAI (11.3%)

DAI (11.3%)

FAI (11.3%)

GAI (11.3%)

BAI (11.3%)



Study Guide for Chapter 5

After you have finished this chapter, you should be able to answer the following questions:

1. What is data abstraction? What is the advantage of using an abstraction?
2. Scalar types describe objects which can be expressed by a single value. What are two scalar types in Ada?
3. How are composite types different from scalar types?
4. When a type is declared, no object is created until the object is declared and given a name, What does this mean?
5. INTEGER is a predefined type in package STANDARD. This covers a lot of whole numbers from -32768 to +32767. How can you limit the range of an object type INTEGER?
6. What is explicit type conversion? Why would you need to do this?
7. What are the two categories of real types in Ada?
8. Why should a programmer declare a specific floating point type?
9. What are the attributes of floating point types?
10. ENUMERATION types are wonderful -- unique to Ada in the many ways they can be used. What can be used as values of ENUMERATION type?
11. What are the attributes of ENUMERATION types?
12. Special Input/Output templates may be used to create new I/O packages for special types you can create. Pay special attention to procedure COMPUTE_DAY_NUMBER, page 181, to see how these work.
13. What is a subtype? Is it a new type? How are they useful?
14. An array is a composite type. What is meant by constrained array types?
15. Procedure INVESTIGATE_MEASUREMENTS, page 189, shows how to load an array, do calculations with the components and display particular components of the array. Be sure you can trace each statement of this procedure.
16. What is an array aggregate? How do you initialize array variables? When are they declared? Procedure LC_LETTER, page 196, is a good example of this.

Programming in Ada

Quiz #1

1. What makes the Ada programming language different from other high level languages?
2. Write a procedure that will display the following on the screen.

```
I'm sick and tired of this machine;  
I wish I could sell it.  
It never does what I want,  
But only what I tell it.
```

Quiz #7

1. What is an unconstrained array? Give an example. Why are unconstrained arrays an asset in data abstraction?
2. Write a procedure to set up an array with an index of colors: RED, GREEN, BLUE, YELLOW, ORANGE, and PURPLE. Step through the array and assign an appropriate flower to each color component.

USE OF PROGRAMS AND PROJECTS TO ENHANCE A SOFTWARE ENGINEERING USING ADA COURSE

Dr. Robert C. Mens
Department of Mathematics and Computer Science
North Carolina A. & T. State University
Greensboro, NC 27411

I. INTRODUCTION:

The author has taught a Software Engineering Ada Language course at North Carolina A. & T. State University every semester since 1985. During this time the course has evolved from an Programming in Ada course with a complete Pascal programming course as a prerequisite to an upper division undergraduate Software Engineering Using Ada course with both Pascal and Data Structures courses as prerequisites. The environment has improved dramatically from several versions of the New York University Ada Ed Compiler to the current production level Digital VAX Ada Compiler. Several programs and a team project embodying major software engineering principles are required in this upgraded course. The primary purposes of this paper are (1) to demonstrate how the programming assignments build upon each other and are used to teach software engineering and Ada Language features and (2) to illustrate successfully completed projects and their use of software engineering principles. In addition, issues such as the use of textbook and resources, available, and the environment are discussed.

II. CONTEXT, ENVIRONMENT, AND RESOURCES:

This Software Engineering Using Ada course is primarily taken by senior computer science majors as a computer science elective. Most of the students graduate and are employed in government and industry within a year of taking the course, and some are doing Ada language work for employers such as Honeywell, DoD, and IBM. Therefore, emphasis is given to preparing these students for the employment environment. The instructor assumes that these students have the maturity to adapt to a new language and develop non-trivial programs.

The environment is the production level Digital VAX Ada System. Because of the need to cover the major features of Ada and software engineering in one semester, such features as run time libraries, library management, the Language Sensitive Editor, and the debugger are not emphasized. However, students are encouraged to explore and use these features in their team projects if they so desire.

The author feels that a textbook does not exist that adequately covers both software engineering features and Ada Language features, and that both an Ada syntax text and a software engineering text are needed. But at a minority school such as North Carolina A. & T., it is generally true that students cannot afford more than one textbook per course. Therefore, only one text per course is required. For several years the text An Introduction to Ada by S.J. Young (7) was used, and the author

still feels that this is the best Ada text from the viewpoint of completeness, readability, and quality of applications.

However, the issue remains as to whether it is better to have a complete Ada Language text and supplement it with lecture notes on software engineering, or to have a complete software engineering text and supplement it with notes on syntax? Since standards and software engineering principles are so strongly emphasized in the work environment, and syntax is so much easier to absorb than software engineering ideas, the author strongly supports the second alternative and has recently adopted the text Software Engineering with Ada by Grady Booch [1]. Despite its difficulty for first time Ada students, those completing the course have reacted favorably to this text once they got comfortable with Ada syntax. In fall 1988 the Watt text [6] was used. The author felt that it was well written from the viewpoint of design and methodology but did not explicitly emphasize software engineering. The students overwhelmingly disliked this text.

In addition, students are encouraged to use the Ada Language Reference Manual [5] and are given the opportunity to purchase or borrow copies.

III. USE OF PROGRAMMING ASSIGNMENTS:

Use of programming assignments is the key methodology used to teach software engineering principles and Ada Language features in depth. Even at a senior level, students need this hands-on experience. However, the students are at the level of maturity that a few (3 or 4) non-trivial programs with multiple objectives is preferable to many short more shallow programs. Even so, the programs are dependent on each other, and reusability, abstraction, and modular design are emphasized to more easily modify and upgrade code.

Objectives of the first program assignment are familiarity with (1) the major features of TEXT_10, including necessary instantiations; (2) subprograms in top down design; (3) the major control structures of the language; and (4) the scalar types of the language. Use of instructor's packages, requirement of student designed packages, and no package requirement have all been tried for program 1. The best students have no trouble designing their own packages, but the slower ones have trouble. In any case, user-defined packages are required in the second programming assignment.

The second programming assignment builds on the first, but also requires use of arrays created at run time and simple records. An external sort procedure using array attributes is needed to sort an array of records on a field such as name or income. Students must work with a minimum of four compilation units (a package specification, package body, main procedure, and sort).

Effective use of generics and exception handling are objectives of the third programming exercise. Here user interface and interactive IO are emphasized, for this will be needed in the team project. From

program 2, the sort is made generic. The program is interactive, providing user prompts and exception handlers to anticipate error in user input. Students are challenged to improve the design of program 2 to support modularity and abstraction, thus increasing the number of compilation units.

The unifying problem for spring 1989 was processing a list of student records where Name, Level (Graduate or Undergrad), Hours_Passed, Hours_Taken, and Cumulative_Points are given. In program 1, the student computed the GPA, Class (FR, SO, JR, SR), and Probation_Status (Boolean) using instructor's criteria. The list is displayed in tabular form. In program 2, the list is treated as an unconstrained array of records and is sorted by name using the enhancements mentioned above. The sort is made generic in program 3. The programmer provides menu driven options that enable the user to choose a sort field. Also, the programmer allows exception handlers for erroneous input of data so that execution can continue.

III. THE TEAM PROJECT:

About one month before the term's end, the students begin working in groups of two or three on a team project. The project's length is about twice as long as a program assignment. The project must involve extensive use of software engineering principles such as data abstraction and information hiding, modularity and separate compilation, and robustness and exception handling. The project must be demonstrated interactively during its own presentation, hence must be user-friendly. Students may select their own topics subject to the instructor's approval, or choose a topic suggested by the instructor. Although not required, use of generics and private types is strongly encouraged. Work with access types, tasking, or discriminated records is encouraged on the project since these areas are not emphasized in the program assignments.

Many of the projects are implementations of abstract data types such as priority queues, trees, and doubly or circularly linked lists. Also popular are data base/file processing and numerical analysis areas. The most outstanding projects of this past year include the following examples.

1) A File Manager for a Small Furniture Company. Operations include insertion and retrieval of records, updating of record components, retrieval of all records with a given attribute, and sorting on any attribute. This was the topic of the most outstanding project done to date, and the project was presented as a student paper at the annual H. C. T. University Student ACM Conference [2].

2) A Financial Transaction Package. A main menu allows users 2 options, insertion of a customer record, deletion of a record, and transaction. The transaction option leads to eight more options. Interest and delinquent charges are made, and records are updated.

3) Infix Expression Evaluator. This is a user-friendly program

that prompts the user to input a parenthesized expression. Messages are given indicating invalid expressions, and valid ones are computed.

(4) Triangle Solver. This program uses a Math Library and solves all possible triangles, finding the 3 unknown quantities when user inputs 3 of the 6 possible values for sides and angles. Exceptions are raised for unsolvable triangles, and multiple solutions are handled.

(5) Generic Numerical Analysis Package. This project uses such methods as Bisection, Newton's Method, and Linear Interpolation to find roots of functions and handles exceptions for no root in an interval and the algorithm failing to converge to a root.

V. CONCLUSION:

The author continues to upgrade and improve this course. This paper is in a sense a continuation of curriculum design in Ada and Software Engineering Education described in [3] and [4]. Future endeavors will include stronger emphasis on object-oriented design and parallel processing. High quality support courseware, especially economical hands-on Ada syntax instruction, would greatly enhance instruction, as would reusable components such as those by Booch. These would be obtained when the University's resources permit it. Possibly an advanced course emphasizing Direct and Sequential IO, in depth parallel processing, machine dependent features, application of reusable components and libraries, and various design methodologies, having the current course as a prerequisite, will be developed.

REFERENCES:

- [1] Grady Booch, Software Engineering with Ada, 2nd edition, Benjamin Cummings, 1986.
- [2] Tony Brokenborough and Mike Ellis, "Use of Software Engineering Principles in a File Management System for a Furniture Company", Proceedings of the 3rd Annual North Carolina A. & T. ACM Conference, 1989.
- [3] Robert C. Mers, "Experiences of Pascal Trained Students in an Introductory Ada Course", Proceedings of the 4th Annual Conference on Ada Technology, 1986.
- [4] Robert C. Mers, "Teaching Software Engineering Principles in a First Ada Course", Proceedings of the 2nd Annual ASEET Symposium, 1987.
- [5] Reference Manual for the Ada Programming Language, Department of Defense, 1983.
- [6] David Watt, Brian Wichmann, and William Findlay, Ada Language and Methodology, Prentice Hall International, 1987.
- [7] S. J. Young, An Introduction to Ada, 2nd Edition, John Wiley, 1984.

**Software Design with Ada
A Vehicle for Ada Instruction**

Orville E. Wheeler
Herff College of Engineering
Memphis State University

The Ada programming language is being introduced in the Electrical Engineering Department at Memphis State University through a course incorporating both the language and some generic design concepts. The intent of combining the two is to provide motivation for mastering the details of the language within a context that illustrates practical applications rather than a typical textbook setting of elementary programming tasks. This paper examines the setting for the course offering in terms of student profile, available equipment, available software, available text resources, and goals. It presents the choices made among the resources and the rationale for them.

The Environment

Memphis State University is a comprehensive university in an urban setting. It is Tennessee's second largest university and enrolls over 20,000 students each term, with about 15,000 FTE. It is located in the largest city and metropolitan area of the region which is a center of agri-business and transportation. The University is the descendant of the West Tennessee Normal School (founded in 1912) and for the first half of its life (until about 1950) focused on producing teachers for the public schools of West Tennessee. It was designated a university in 1957 and has been broadening its scope since then. The Herff College of Engineering is a relatively new (25 years) engineering program that in 1987 added the Ph.D. degree to its offerings. Even though there is not an abundance of high technology industry in the immediate area to provide high visibility for engineering as a career, the population concentration results in a stable and moderate enrollment of about 1800 students, or roughly 1100 FTE. The College offers baccalaureate, master's, and doctoral programs in Civil, Electrical, and Mechanical Engineering along with a baccalaureate program in Engineering Technology, a master's program in Industrial and Systems Engineering, and a master's and doctoral program in Biomedical Engineering. As with most engineering institutions, the EE's have the largest enrollments. The circumstances and characteristics of the institution are readily recognizable.

While there has been a modest level of research and interaction with various research funding agencies for over a decade, intense interaction, particularly with the Department of Defense and some of its primary contractors, began only about five years ago through the efforts of three members of the Electrical Engineering Department. Those efforts have raised the level of awareness in the College of the need to provide our students with the opportunity to become acquainted with Ada, and, along with a growing recognition of the importance of Ada outside of the government procurement system, convinced us that Ada must be included in our curriculum. The Computer Science degree program at Memphis State is housed in the Mathematical Sciences Department, and the only course offering in their curriculum which touches on Ada is a comparative course contrasting Pascal, Ada, and C. It was felt that a course in the Electrical Engineering Department, which offers a number of

other computer hardware and software courses, was an appropriate vehicle for introducing Ada into the curriculum in the College.

The target students for this course, at least initially, were electrical engineering seniors (although other engineering majors and computer science majors are expected to enroll in subsequent offerings). These students (the EE's) have all had an introductory course in Pascal with some exposure to Fortran, and they have had a matrix computer methods course that requires extensive programming in Fortran. Some have had a course utilizing Unix and C as an elective in addition. They are not neophytes, and their level of sophistication concerning computer hardware and software is good.

The Specification

The Accrediting Board for Engineering and Technology (ABET), the accrediting agency for engineering programs, has a uniform requirement of one half year of instruction in design for all engineering programs. Design is very hard to teach, and any opportunity to include some design content in an upper division course is usually seized. This results in courses being credited with some portion of the total credit being allocated to the design requirement whenever it is appropriate. Courses taught for the purpose of introducing the student to a programming language are considered skills courses and receive little regard from ABET, particularly at the senior level. In formulating the first Ada course in our college, I felt it was essential to include a design focus to make it attractive as an elective that would be recommended by faculty advisors to our students. This, along with the Ada's facility for large system design, made a course titled "Software Design with Ada," a natural choice. The intent was that the course cover design procedures with the student learning to work in Ada as the instrument for embodying design decisions. The primary goal of the course is to develop the ability in the student to successfully attack a software design task using Ada. This requires, probably more than other languages, the ability to use a large and diverse body of reference material.

A secondary goal of the course was to develop the capacity in the student to work effectively in the computer environment rather than relying exclusively on paper. An unusual aspect of this course (at least for us) was that it was conducted entirely in electronic media, including quizzes and exams. Paper was a secondary medium and students were not allowed to submit material on paper.

The Tools

The resources necessary for teaching Ada, or any other computer language, can be grouped into four categories: faculty, hardware, software, and computer time. The last is the easiest to acquire at Memphis State: computer time and access to the mainframe is provided at no cost to students enrolled in a course in which it can be used. There are also several (three in the College) microcomputer labs that are open to all students and many, though by no means all, of our students own their own computers. Faculty is a little more difficult; Ada is not widely known among the faculty and there is little motivation to learn and use anything but Fortran. Some of the EE faculty have picked up C and they use Pascal for instruction, but, as in most engineering

environments, Fortran is the computer mother tongue. I was self selected to teach the first section because I believe it is important. I have studied the language for several years, and since I acquired a compiler a few years ago and have been using it, I was the only faculty member in the College who had programmed in Ada.

Hardware is not a major problem at Memphis State either. Equipment available for this course at the time it was planned included a broad array of processors and operating systems, ranging from Zenith 159's (IBM PC-XT compatibles) running MS-DOS, through AT&T 3B2-400's running Unix, and a Prime 750 minicomputer running Primos, to a Univac 1100/82 running the current version of the EXEC operating system. The hardware was not a constraining factor. (Since the initial class started, a VAX-8820, with VMS, has come up and is available to students.)

At the time the first section of the course was being planned and scheduled, equipment and software vendors were surveyed to determine what was available for instruction in Ada. Three vendors were identified for validated Ada compilers on the micros: Meridian, Alsys, and R&R Software (Janus/Ada). Bell Labs has a validated Ada compiler for the 3B2-400 and Prime has a validated Ada compiler for some of its newer Series 50 products. The Univac system was scheduled for replacement so the price of the Ada compiler for a short time use was prohibitive. The VAX had been selected as a replacement for the Univac and of course, DEC has a VAX Ada compiler running under VMS on its systems. There are several of the AT&T 3B2 machines available in the Electrical Engineering Department with several terminals attached to each, so they were a strong candidate for selection. I pursued acquisition of the AT&T Ada compiler, first through the local AT&T representative, then the regional representative and finally directly with Bell Labs personnel. No one short of the Labs could even get any information on the AT&T Ada compiler. The person I spoke with at Bell Labs acknowledged that they had the validated compiler but would not sell or lease it. The argument was that that one belonged to the government and they were working on another one which they would eventually market. It was clear that we were not going to get their compiler.

Prime has a validated compiler for its Series 50 machines but they have not validated it on the Prime 750 since it is a discontinued machine. They suggested that we upgrade to one of their newer machines and buy the compiler for it. That left the Univac, which was to be replaced, the VAX, which was not yet on the scene, and the micros. I had been using the Alsys compiler for MS-DOS machines for some time and was familiar with it. I had used the unvalidated Janus/Ada compiler also but I had not used the Meridian compiler. RR Software had succeeded in getting their compiler validated so I bought a copy of their validated ED-PAK and worked with it for a while. I felt the Janus/Ada ED-PAK was satisfactory, and the price was certainly right, so it was selected for use on our campus. (The unrestricted site license for the Janus/Ada ED-PAK cost approximately what one copy of the Alsys compiler costs. One should note that the Alsys compiler came with a required add-in board with four megabytes of additional memory.) Meridian lost out to what we considered to be a commodity price for the Janus/Ada system, without even a cursory review. Thus we came to the use of the Zenith 159 computer, our lab machines, with the Janus/Ada ED-PAK¹.

Having the hardware and software selected, the next task was the selection of a textbook and related reference materials. About a dozen books were considered as a primary text, ranging all the way from one intended to be an introduction to programming², to lengthy comprehensive language treatments³. (Only those based on the 1983 ANSI/MIL Standard were seriously considered. There are still a surprising number based on draft versions of that standard, e.g., Habermann⁴). *Software Engineering with Ada* by Grady Booch⁵, was selected on the basis of its mix of Ada instruction with object oriented design guidelines. Of the books reviewed, this is probably not the best book on the language, but it does fit the intent of the course well. (My personal choice for the best book on the language is Cohen's *Ada as a Second Language*³.) Additional material on the language, including the ANSI/MIL-STD-1815A⁶, is used to supplement this text, along with some of the extensive literature on software engineering and design by people like Boehm⁷, Yourdan⁸, and Brooks⁹, and more recent work like that of Pressman¹⁰ and Somerville¹¹. Reference material on design in general relies on published work such as Alexander's *Notes on the Synthesis of Form*¹², the works of Archer¹³, Rittel¹⁴, and particularly Bazjanac's¹⁵ idea of design as a learning process in *Basic Questions of Design Theory*¹⁶.

The Implementation

The course is organized to cover all of the primary features of Ada in about three quarters of the semester with design lectures mixed in. It happened that the course was taught for the first time with a Tuesday-Thursday schedule with 85 minute lecture periods each day. The coverage of the material in Booch's book, which is reasonably complete if sometimes brief, along with several lectures on design and three quizzes, occupied 21 lecture periods. The remaining six meetings were devoted to designing, developing, and implementing a solution to a particular design problem as a group project, with the teacher as a participant. It should be noted that the course intent is software design, not software engineering. That is, system specification on one end of the life cycle and extensive testing and maintenance on the other end are mentioned only briefly in establishing the context in which the software design and implementation occur. Since it is evident that one learns a language, computer or otherwise, by using it, a heavy schedule of outside programming assignments accompanied the reading assignments, 14 in all in addition to the project.

The general design content of the course relied on the following definition:

"Design is the economical allocation of available resources to meet a perceived need."

The presentation developed generic design ideas following historical lines in this very brief form. (This material follows the paper by Bazjanac to some extent.) Many people, including Booch, use a direct linear model for the design process. Since this model (a linear series of steps beginning with problem identification) has been around so long (it was implicitly in the work of Vitruvius) it is ingrained in the literature on design. It has been the accepted model of design until the last half of this century. Even now, some contemporary models are based on it. This system, sequential steps in a

predetermined process, forms the basis of what are called "first generation" design theories. The precise steps to be taken are varied from model to model but the basic idea of a linear sequence, and since World War II, the inclusion of operational research techniques under the title of optimization, remains fairly static.

One modified form of it (popularized by Alexander¹²) calls for two basic steps which he labels "analysis" and "synthesis." The first is an analysis of the problem to be solved by the design and the second, the formulation of the solution. The key point is the separation of the understanding of the problem from the development of the solution. First one, then the other. This appeals very much to the systematic orderly mind of an engineer. At the time Alexander published his book, he believed that design problems could be defined in a hierarchical way, and that they could be broken down into simpler subproblems for solution. This is the same presumption built into the idea of top down structured program development which is relatively popular now in software design. It is an independent version of the formulation made popular by Dijkstra¹⁷.

Another relatively recent (1963) formulation, by Archer¹³, broadened this a little to include three phases: Analytical, consisting of observation, measurement, and inductive reasoning (define the problem); Creative, consisting of evaluation, judgment, deductive reasoning and decision-making (synthesize the solution); and Executive, consisting of description, translation and transmission (communicate the solution). This last phase goes beyond many author's last step by also listing communication after implementation as part of design. Archer also modified the model by noting that feedback occurred between steps. This can be seen in software development literature in the work of Boehm in describing the software life cycle.

Even this model with its single step feedback isn't adequate for many design tasks and the reason is that many design problems are in a class defined by Rittel¹⁴ and subsequently labeled as "wicked" by another writer. Wicked problems have eleven properties which set them apart from well formed design problems. Unfortunately, many, if not most, software design problems have some or all of these properties. The eleven need not be elaborated here but a quick summary shows the significance for software design. They are:

1. Wicked problems have no definitive formulation.
2. Every formulation of the wicked problem corresponds to the formulation of the solution (and vice versa).
3. Wicked problems have no stopping rule.
4. Solutions to a wicked problem cannot be correct or incorrect. They can only be "good" or "bad."
5. In solving a wicked problem there is no exhaustive list of admissible operations. Anything is permissible in finding a solution and nothing is mandatory.
6. For every wicked problem there is more than one solution. The selection of an appropriate solution depends on one's point of view.
7. Every wicked problem is a symptom of another "higher level" problem.
8. No wicked problem and no solution has a definitive test. No matter how one tests, another set of input may cause failure.
9. Each wicked problem is a one shot operation. There is no room for trial and error, and there is no possibility of experimentation.

10. Every wicked problem is unique.

11. The wicked problem solver has no right to be wrong. It must be right the first time.

These properties invalidate the linear sequence of decision models. It is clear that some other paradigm of design is necessary for them. (Note that small programming tasks, such as those usually used in teaching programming and programming languages, indeed that are looked at in any way in instruction, are nearly invariably well formed. Large problems are highly unlikely to be though.)

Rittel proposed a model to attack wicked problems which is too lengthy to present here but is summarized in Bazjanac's paper. Rittel's formulation is general enough that there can be, indeed must be, feedback from all steps to all others. The context changes the model which changes the performance, etc. The model interacts with the context so that some features are important in one model and not in another, etc. He describes the process as:

"...The designer is arguing toward a solution with himself and with other parties involved in the project. He builds a case leading to a better understanding of what is to be accomplished."

He also says "designing means thinking before acting."

The final installment of this long passage is the idea of design as a learning process, which was put forward by Bazjanac himself. Bazjanac says that the crucial point is to recognize that one may determine a formulation of the problem and work on its solution while all the time keeping in mind that the problem formulation is not final, that it is subject to change. At any time the designer is working on the best solution he can based on the knowledge he has of the problem at that point. As the solution becomes more and more definite, the problem statement will change to reflect the designer's increased knowledge of the problem. As the designer learns more about the problem, he recycles the loop of analysis and synthesis. If he really learned something significant, the solution will change; if not, it will stay the same. Note that this process provides an automatic stopping rule even for wicked problems. When nothing significant changes in a cycle, it's time to stop. Usually, however, the stopping rule turns out to be elapsed time.

This concept of design provides one explanation of why design is so hard to teach. The ability to design, in this view, depends very heavily on the accumulated experience of the designer. The things he does in learning about a particular design problem make use of the things he has done before and the insights gained in previous design efforts. There are many things he can do because "I've done it before and it works." This is not communicable in a rote learning context. It is an often repeated statement, but it is true, you learn to design by designing; it is of necessity on-the-job training. (Note that this reduces well formed design problems to something less than "real" design problems.)

The material used to present the semantics and syntax of Ada are no different than that used by many other people so the remaining point of interest is the class project. (It might be noted in passing that teaching Ada is a wicked problem in the sense of the characteristics enumerated above.)

The selection of a design project was made after discussions with several members of the EE faculty and an extended discussion with the students in the class. It was felt that a single project with different parts being developed by different students and then the whole integrated would be possible and instructive. It was felt that something with utility beyond this class was to be preferred. The problem finally adopted was taken from the current research work of a doctoral student in the management sciences program in the College of Business. His work provides an independent check on the results and may be extended through expansion of the work done on this project.

The object of study was a network with nodes made up of servers with queues. The project was to design and implement a simulation of this system in Ada which could be used for expansion to more complicated networks (in this project, the queue was provided but the queue length was one for simplicity). The model was to have an infinite population that provided transactions entering the system at an exponentially distributed random time interval with a mean time λ . These transactions were to be served at the first node, again with an exponentially distributed random time but with a mean time μ . After being served at node one, the transaction moves to node two, if it is not busy, or node three if two is busy, or waits if both are busy. While a transaction is waiting or being served at node one, any other arriving transaction is blocked and balks out of the system. A transaction arriving at node two or node three is served, as in node one, with an exponentially distributed random time with mean time μ , and then moved out of the system. The objective of the simulation is to gather statistics on the network performance including: the probability of any single transaction balking, the probability of there being zero, one, two, or three transactions in the system at any given time, the average number of transaction in the system at any time, and the transit and wait time for an average transaction. The probabilities that are the principal interest in this investigation are functions of the ratio of λ to μ .

The tasking facility of Ada is a natural framework for this kind of problem. After a lecture from one of our management sciences professors to introduce the students to the idea of the network and the necessary probability and statistics, Booch's standard litany of design steps was followed with various students selecting various parts of the problem to attack. In two and a half weeks, a system, containing 13 user packages, three containing tasks, was producing answers, slightly incorrect, but answers nevertheless. One of the packages, a random number generator, was taken from the Ada Software Repository distributed by Advanced Software Technology, Inc., and slightly modified to make it run correctly on a sixteen bit machine. A generic queue package was adapted from *Software Components with Ada* by Booch¹⁸. I wrote two small and one substantive package (none of the tasking), and the balance and all of the documentation were written by the students.

The Results

The first offering of this course has been completed and some observations can be made. The objectives of the course are certainly within reach when the students have had a moderate exposure to higher level languages. Seven students initially registered in the course and six successfully completed it. One (a nominally good student with a 3.25 GPA) dropped the course after missing several lectures and apparently deciding the work load

exceeded the benefit he would derive from it. This small class is, of course, a luxury when introducing a new course that requires some experimenting to establish lecture content and assignment work loads.

Janus/Ada and an acceptable full screen editor were loaded on every machine in one of our labs containing Zenith 159's, and there was never any waiting to get at a keyboard. (I went to all this trouble before I knew how many people would register for the course.) This level of computing power is low however, and the students in the class, without exception, moved very quickly to the VAX when it became available about six weeks into the course. The rules for turning in material and giving quizzes were not changed though, and all material was brought back to micros and turned in on floppy disks. All quizzes and the final exam were conducted on the micros. This makes a strong statement about the relative attractiveness of working with Ada on the micros and the VAX because there was absolutely no help given to the students on the VAX. The accounts were opened and the students had to learn enough about VMS and running Ada on the VAX on their own to do the work. It is true that students refer to each other and to preceding classes for this sort of thing very effectively, but in this case there was no preceding class; the VAX, VMS, and Ada, were new to our campus.

During the project phase of the term, the students were talking and thinking about the problem and not about the features of Ada required to implement it. This indicates that the basic semantics and syntax of the language were in hand as a result of the heavy outside assignments. The only feature of Ada to get a lot of discussion at this time was tasking.

Student comments, solicited in an informal office session, at the end of the project phase, provided some recommendations for the next offering of the course. Booch's book was criticized for not having enough small examples that highlight specific aspects of the language. Those familiar with the book will recall that it relies on a few relatively large (for instruction) systems, each of which contains many previously unused features of the language. The inadequacy of the index in Booch's book was also noted. A search for a better book that meets the universal student's ideal for textbooks, both short and absolutely complete, was recommended. Adherence to the use of the micro in the face of the clear superiority of the VAX was very unpopular. Surprisingly, the difficulty of the project did not result in criticism. I have long felt that we do not challenge our students enough in our curriculum, but I believe this project did stretch their capabilities, and they recognized it. Two of the students continued to work on the model after the semester ended.

My reflections on the course are a little different. I think this first effort has been successful but can be improved. A more careful selection of the class project might be made to expose more of the facility of Ada. Some use of generics and access variables might have been more productive than so much work with tasking. I am convinced, although none of my faculty colleagues are, that Ada can be used for an introductory course in programming with the students in our college, with exactly the same benefits as Pascal. It would not be possible to cover everything in the language but at least as much as is covered in a course in Pascal could be presented. At the senior level, it would be impossible to devote enough time to one course for the instructor to grade and debug for the typical class of 25 to 30 as deeply as

was possible with this class of six. One senior EE faculty member sat in on the course so there will be more faculty available in the future.

I enjoyed the course very much and I expect Ada to spread in our College.

References

- [1] Stock, D. L., et al, *JANUS/Ada Compiler User Manual*, RR Software, Inc., Madison, WI, 1988.
- [2] Mayoh, B., *Problem Solving with Ada*, John Wiley & Sons, New York, NY, 1982.
- [3] Cohen, N. H., *Ada as a Second Language*, McGraw-Hill Book Company, New York, NY, 1986.
- [4] Habermann, A. N. and Perry, D. E., *Ada for Experienced Programmers*, Addison-Wesley Publishing Company, Reading, MA, 1983.
- [5] Booch, G., *Software Engineering with Ada*, Second Edition, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1986.
- [6] -----, *Reference Manual for the Ada Programming Language*, United States Department of Defense, ANSI/MIL-STD-1815A, 1983.
- [7] Boehm, B. W., "Software Engineering," *IEEE Transactions on Software Engineering*, January 1977, Vol. SE-3, No. 1, Reprinted in *Classics in Software Engineering*, E. N. Yourdan, ed. Yourdan Press, New York, NY, 1979.
- [8] Yourdan, E. N., ed., *Classics in Software Engineering*, Yourdan Press, New York, NY, 1979.
- [9] Brooks, F. P., Jr., *The Mythical Man-Month Essays on Software Engineering*, Addison-Wesley Publishing Company, Reading, MA, 1975, Reprinted with corrections, 1982.
- [10] Pressman, R. S., *Software Engineering A Practitioner's Approach*, McGraw-Hill Book Company, New York, NY, 1982.
- [11] Sommerville, I. and Morrison, R., *Software Development with Ada*, Addison-Wesley Publishing Company, Reading, MA, 1987.
- [12] Alexander, C., *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.
- [13] Archer, L. B., "Systematic Method for Designers," *Design* No. 172-188, 1963.
- [14] Rittel, H. W. J., "Some Principles for the Design of an Educational System for Design," *Journal of Architectural Education*, Vol. XXVI, 1971.

[15] Bazjanac, V., "Architectural Design Theory: Models of the Design Process," in *Basic Questions of Design Theory*, W. R. Spillers, ed., American Elsevier Publishing Co. Inc., New York, NY, 1974.

[16] Spillers, W. R., ed., *Basic Questions of Design Theory*, American Elsevier Publishing Co. Inc., New York, NY, 1974.

[17] Dijkstra, E., "Programming Considered as a Human Activity," *Proceedings of the 1965 IFIP Congress*, North-Holland Publishing Co., 1965, Reprinted in *Classics in Software Engineering*, E. N. Yourdan, ed., Yourdan Press, New York, NY, 1979.

[18] Booch, G., *Software Components with Ada: Structures, Tools, and Subsystems*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

TRANSITIONING TO ENGINEERED Ada IN THE SMALL LIBERAL ARTS COLLEGE

Ronald H. Klauswitz

West Virginia Wesleyan College

INTRODUCTION

West Virginia Wesleyan College has an enrollment of about 1400 students. About 40 are computer science majors. The curriculum was introduced in 1975 under the ACM and IEEE guidelines. It was decided by the faculty in 1987 that a transition would be made from the Pascal language foundation to an Ada foundation to take place over a period of three years. The process was deliberately slow with options for bailout at any point along the line until the conversion of the CSC 1 course was complete. The reasons for that conversion and its results to date are the subjects of discussion for the rest of this paper.

THE DECISION

Any kind of transition makes extra work. Pascal is block structured and allows us to teach good programming techniques. Instructors have all chosen our favorite texts (from many available) and have class notes, tests, syllabi, and overheads that are tried and tested. Why change?

The problem, for one thing, is that Pascal has never really been accepted by the real world. Students who take jobs in the D.C. area (which is the majority of our graduates) are coming back with reports that interviewers are asking for Ada credentials early in the process. It seems much better to train students in a language that they can actually use than to train them in one they should use as a template or example.

Secondly, we are located in an area of West Virginia that Senator Byrd is trying to convert from a depleted resource-extraction base to less destructive and more lucrative hi-tech. The backbone of that project is the teaching of the Ada programming language in the colleges of West Virginia.

The last reason involves the quality of the Ada language itself. Through the initial rain of criticism there is emerging an opinion that Ada does have some fine qualities. In the areas of syntactic orthogonality, machine independence, extensibility, abstraction and encapsulation, and adoption of software engineering techniques Ada compares favorably with Modula-2 [2].

Enter Engineering

The topic of software engineering brings up a question as to what students are really being taught in first-year programming courses. They are given a lot of programming exercises to do which they hastily throw together to look just good enough and work just well enough to get the grade. Then the programs are thrown into the trash and the process begins again. How can engineered features such as modularity or reliability be taught in programs that are one-time, run and discard? In contrast Ada is the essence of building-block programming. Carefully thought out exercises would let the student build a simple module, then use it to construct a more complex module and so forth. By the end of the semester all of his/her sins or virtues are there (working or not) in a huge project.

There can be no argument to the importance of software engineering in a curriculum. Denning [3] has included it as one of the nine factors in a definition matrix for the discipline. To become part of the students thinking it must be stressed from the beginning forward.

IMPLEMENTATION

First, a word of warning. There is a reason that small colleges can compete with large universities. Teaching is done on a more personal basis made possible by smaller classes. So my disclaimer is that the specifics which follow may not be universally applicable. I will be grateful if they even work at Wesleyan.

The Time-line

Our agenda in chronological order follows below.	Year
1) Teach Ada on a trial basis for the experience	1
2) Identify courses which must change	1
3) Train instructors	1
4) Select books	1
5) Write syllabi	1
6) Convert CS 1	2
7) Convert CS 2	3

Trial Teaching

For Wesleyan, the trial teaching was easily accommodated. We have a January short term that is set up for just this kind of experiment. The Ada course was a great success. It turned into a lecture/lab combination at the request of the students. Their reasoning was that when you are stuck with a diagnostic in Ada it is more difficult to diagnose and correct the problem than in other languages. This led to frustration if quality help was not available. The lesson was not lost for designing the CS 1 course.

AD-A214 663

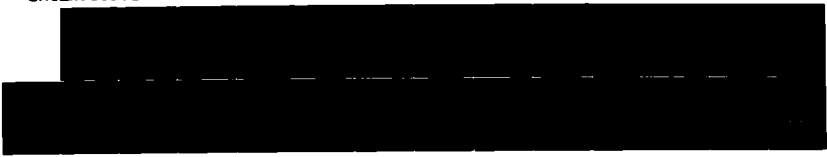
PROCEEDINGS OF THE ANNUAL ADA SOFTWARE ENGINEERING
EDUCATION AND TRAINING... (U) ADA JOINT PROGRAM OFFICE
ARLINGTON VA JUN 89

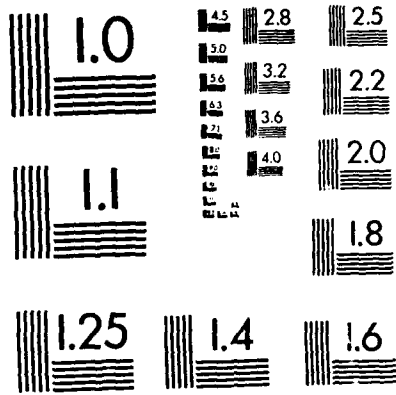
3/3

UNCLASSIFIED

F/G 12/5

ML





The Courses

The higher level courses, operating systems, programming languages, numerical methods, database systems, and data structures are generally language independent. It is in CS 1 and CS 2 that most of the changes happen. I will consider these separately.

CS 1

Ada is generally expected to require a prerequisite. The structures and concepts in Ada may be very difficult for a first language. The recognized prerequisite is Pascal. There is, however, some controversy in this. Since Pascal is so close to Ada in many areas there is a definite potential for confusion when both are introduced back-to-back. There may be a better transition from other languages. Regardless of the answer to this question, we have no room in our curriculum for another course. The prerequisite question then must be dealt with in other ways.

There is another consideration. Wesleyan, like most institutions, has seen a decline in Computer Science freshmen over the past couple of years, but it has been accompanied by a greater staying power in the students who do enroll. These students have been exposed in high school to enough programming and computer experience to dissolve a lot of the romance and to give the program entrants a good background. But, that can not be accepted on a blanket basis. We have got to go on a case-by-case basis. And this is where the "SMALL COLLEGE" part comes in. We ask declared majors to fill out a background form. Further, we will be giving an entrance test for CS 1. I have used a similar test every year for about six years in the first language courses, BASIC and FORTRAN. We will be supplying support people, faculty or seniors on workstudy, to work one-on-one with people targeted by the test. Another very important part of the process will be a large semantics section at the beginning of the class just trying to get to the point where instruction can begin on a more even footing.

CS 2

It will be difficult in this course to find a book that successfully covers the classic CS 2 as it should [4]. A brief description of our CSC 2 course follows.

1. An intense coverage of the Ada language including the following items:

- Ada PACKAGES
- GENERIC PACKAGES
- SEQUENTIAL FILE I/O
- DIRECT FILE I/O
- DATA STRUCTURES
- SORT AND MERGE CONCEPTS

2. A good introduction into the concepts of logic and logic circuit combinations including:
 - LOGIC CIRCUITS
 - BASIC CIRCUITRY FOR GATES AND STORAGE
 - STATE MACHINES
 - BOOLEAN LOGIC
 - LOGIC REDUCTION
 - INTRODUCTION TO ARCHITECTURE
3. Instead of "throw-away" programs we will use a "build-up" concept where early exercises are used as work already done and useable in later projects.
4. Exercises that tie hardware and software together will be used throughout the course. For example, a program that could do simple logic reduction has been an exercise that has been used in CS 2 for years.

Book Selection

Over thirty-five books were screened in an attempt to find a book for the CS 1 course that was complete and self-explanatory. I highly recommend the AdaIC packets [1] as a place to begin. It contains (among other very useful items) a current review with comment of all Ada textbooks in print. Best of all, it is free.

For the CS 2 course there were no contenders. Hopefully the future holds a good book for this course, but for now we will be using an advanced Ada text with handouts for the logic topics.

CONCLUSIONS

I have presented here a condensed version of many discussions that led up to a decision to transition to the Ada language in the computer science curriculum at West Virginia Wesleyan College. I have additionally discussed the changes, as we see them, that must be made to specific courses to accommodate that change. Without a deliberate and thought-out transition it is very obvious to me that severe problems could be encountered.

REFERENCES

1. AdaIC, General Information, Historical, and Educational Packets., AdaIC, 3D139 (1211 S. Fern, C-107), The Pentagon, Washington, D.C 20301-3081, (703) 685-1477
2. Belkhouche, B., Lawrence, L., and Thadani, M. A Methodical Comparison of Ada and Modula-2. Journal of Pascal, Ada & Modula-2, 7 (July/August 1988), 13 - 24.
3. Denning, P.J., et. al., Computing as a Discipline. Communications of the ACM. 32 (Jan 1989), 9-23.
4. Denning, P., What is computer science? Am. Sci. 73(Jan-Feb.1985), 16-19.

**A HyperCard Prototype of a CASE Tool
in Support of Teaching
the Berard Method of Object-Oriented Development
in an Undergraduate Course**

Frances L. Van Scoy
Department of Statistics and Computer Science
West Virginia University
Morgantown, West Virginia 26505
vanscoy@a.cs.wvu.wvnet.edu

Background

Pascal GENIE, a programming environment targeted at freshman students learning Pascal, has been developed and used at Carnegie Mellon University since 1986 (Chandhok, 1989). This integrated environment is implemented on Macintosh computers and supports procedural abstraction, data abstraction, program assertions, and visualizations.

Now a similar environment for freshman students learning Ada is being developed by an expanded version of the original team, including members from Carnegie Mellon University and West Virginia University. One task within the new project is the design and implementation of student-strength CASE (Computer-Aided Software Engineering) tools, integrated within the environment, which encourage good software engineering practice in Ada. This paper describes early work on a prototype for one tool which might be included in the Ada GENIE.

Ed Berard, a prominent Ada software developer and trainer, has often said, "Ada without a methodology is trash." In agreement with this statement, we intend to provide tools which will encourage the teaching (at the undergraduate level) of one or more software development methods such as OOD (Object-Oriented Development) (Booch, 1987b; Berard, 1985) or PAMELA2 (Pictorial Ada Method for Every Large Application) (Cherry, 1988).

Part of our strategy in developing such tools is to build a prototype tool in support of OOD. We have tested it at WVU, during spring semester, 1989. We intend to modify the prototype, again test it during fall semester,

1989, with students in an Ada-based CS 1 course, and then develop the requirements and specifications for one of the tools to be built as part of the integrated environment during academic year 1989-1990. We will also investigate other tools in support of OOD and PAMELA2.

Overview of Object-Oriented Development

Booch (1987b) writes, "Simply stated, *object-oriented development* is an approach to software design and implementation in which the decomposition of a system is based upon the concept of an object. An *object* is an entity whose behavior is characterized by the operations that it suffers and that it requires of other objects. By *suffers* an operation, we mean that the given operation can legally be performed upon the object."

Two strategies of applying OOD have been published. Grady Booch (1983) described a method that begins with the writing of an English paragraph of five to nine sentences which states a solution to the problem. The noun phrases of the paragraph become the objects, and the verb phrases become the operations in the software solution. Ed Berard (1985) refined and formalized this method. We will refer to this way of doing OOD as the Berard method throughout the remainder of this paper.

Booch (1987b) has also described a method for OOD that begins with identifying inputs and outputs of a system and drawing a data flow diagram for the system. We will refer to this as the Booch method.

Modified Berard Method of OOD

At West Virginia University, we teach a modified version of Berard's method of OOD. The tool described in this paper is designed to support this modified version.

The steps of the modified Berard method are:

1. Write one English sentence which describes the problem to be solved.
2. Gather, analyze, and organize the information needed to solve the problem.
3. Write one English paragraph which describes a solution to the problem.
4. Identify the objects.

- a. Select noun and pronoun phrases in the paragraph.
 - b. Complete the entries in the Object Table (noun phrase, problem space versus solution space, Ada identifier for the object, and indication of which objects are closely related).
 - c. Identify the attributes of each object (actor, server, or agent; type versus object; and phrase or sentence describing purpose of object).
5. Identify the operations.
- a. Select verb phrases in the paragraph.
 - b. Complete the entries in the Operation Table (verb phrase, problem space versus solution space, major object associated with the operation, Ada identifier for the operation).
 - c. Identify the attributes of each operation (constructor, selector, or iterator; function or procedure; phrase or sentence describing purpose of operation; error conditions).
6. Identify the program units.
- a. Group objects, types, and operations.
 - b. Identify the high-level Ada program units (generally library units) and specify their interfaces.
 - c. Draw a Bocch diagram to show the dependencies among program units.
7. Implement the Ada program units.
- a. Write the Ada unit specifications and compile them.
 - b. Write the Ada unit bodies and compile, link, and test them.
8. Repeat the process as necessary: iteratively to correct mistakes, recursively to refine the solution.

Need for the CASE Tool

In practice, OOD is both an iterative and a recursive method. As development progresses, errors in the design are observed and the process iterated to make corrections. After a top-level design has been completed and package specifications written, the process is applied recursively to develop the system at lower levels.

Students often become frustrated as they iterate to correct errors in the design. For example at step 5b a student may observe that an additional operation not suggested by the verbs found in step 5a is needed. Adding a sentence with that verb means a change in the paragraph written at step 3 and likely causes a change in the list of noun phrases of steps 4a, 4b, and

4c. The result of making changes and repeating steps is that students are continually either copying large amounts of material with only a few changes or making small modifications and hoping that all implied changes have been made.

To encourage students to use the method properly, we would like a tool which will automatically copy information from step to step, will keep all information in the system consistent, and will prompt students at each step of the process to enter the needed additional information.

Strategy in Building the CASE Tool

HyperCard was chosen as the system with which to build the prototype tool because it provides a friendly user interface and because HyperCard stacks are easily modified. This ease of modification is helpful in at least two respects. The designer of the tool can easily make changes as the need for them becomes apparent. Also, students who are fluent at using HyperCard stacks will begin using HyperCard features for added functionality. If we can capture the HyperCard shortcuts used by students, we can then add the appropriate functionality to our tool.

Essentially, a HyperCard "program" consists of one or more stacks of cards. Each card may have several fields and several buttons. A field may contain text or graphics. A button has associated with it a script which is executed when the mouse is clicked while pointing to the button. Scripts can be copied from the many sample cards and stacks provided with the HyperCard system. More complex scripts can be written in the HyperTalk language. Danny Goodman has written several books which are useful for HyperCard stack designers and implementers (Goodman, 1987, 1988a, 1988b).

Functionality of our CASE Tool

Our tool assists the student in following the Berard method of OOD. First the student is prompted to enter the required information for steps 1, 2, and 3. Next (step 4a) the student uses the mouse to select noun phrases from the paragraph written in step 3. These phrases are displayed in a skeleton Object Table whose entries the student can then complete (step 4b). The tool builds a stack of cards, one for each object, with all information available from the Object Table. The student can then travel through this Object Stack and add additional attributes for each object (step 4c). Steps 5a, 5b, and 5c are followed in a similar fashion. At the start of step 6a the tool sorts the objects and operations identified

earlier into collections of related objects, types, and operations and allows the student to confirm or edit these groupings. The information approved in step 6a is then used by the tool in step 6b to request visibility information. (Step 6c should draw a Booch diagram for the system but is not fully implemented.) The tool then provides partial Ada unit specifications and bodies in steps 7a and 7b for editing by the student.

Example of Ada System Design using Our Tool

The use of our tool is shown by the following example which is taken from an exercise used in a junior-level elective during spring semester, 1989, at WVU.

Statement of Problem

A few times each year I need to bake a certain number of cookies for a meeting or reception. I'd like to use more than one recipe for the sake of variety.

I'd like a software system which allows me

(1) to record information about each of my cookie recipes.

This information must include:

- (a) yield (the number of cookies produced by one batch of this recipe)
- (b) a list of ingredients with quantity needed of each
- (c) the oven temperature required by this recipe

(2) to select some of the recipes I have recorded and generate a report. The report should contain:

- (a) a list of the chosen recipes
- (b) the total number of cookies produced by preparing these recipes
- (c) the total amount of each ingredient needed

(d) an order for baking the cookies (I want to bake the cookies requiring the lowest temperature first, and the ones requiring the highest temperature last.)

Some issues:

- (1) The amount of an ingredient may be expressed in various units:
ex. 1 pound, 1 stick, 1/2 cup, or 2 tablespoons of butter
- (2) Some ingredients do not have units of measure attached to them.
ex. 3 eggs
- (3) The recorded recipes must persist between uses of the system.
- (4) I want the ability to add more recipes at a later date.
- (5) During a "baking planning session" I want to be able to choose some recipes, view the report, and then add and/or delete recipes from the list of chosen ones.

The first three cards, not shown here, display a welcome message and prompt the user for a single sentence statement of the system to be developed and for references to information needed to solve the problem.

In step 3, the user is asked to enter a paragraph describing a solution to the problem. An example of such a paragraph is shown below.

3. Write one English paragraph which describes a solution to the problem.

Enter recipes into a recipe box. Select several recipes from the recipe box. For this collection of selected recipes determine the total number of cookies to be baked and the total amount of each ingredient.

Step 2

Step 4a

Prepare Design Report

Print Design

In step 4a, the user is given a copy of the paragraph written in step 3. The user repeatedly selects a noun or pronoun phrase and then clicks on the "Select Noun Phrase" button. The script for this button copies the selected phrase into the first column of the Object Table on the card for step 4b and changes all lower case letters in the selected phrase to upper case in the text field on the card for step 4a. When satisfied that all noun and pronoun phrases in the paragraph have been selected, the user clicks on the "Step 4b" button.

4. Identify the objects.

4.a. Select noun and pronoun phrases in the paragraph.

Enter RECIPES into A RECIPE BOX. Select SEVERAL RECIPES from THE RECIPE BOX. For THIS COLLECTION of SELECTED RECIPES determine the TOTAL NUMBER of COOKIES to be baked and THE TOTAL AMOUNT of EACH INGREDIENT.

Select Noun Phrase

Step 3

Step 4b

Prepare Design Report

Print Design

In step 4b the user is given a copy of the Object Table with the Noun column filled with the noun and pronoun phrases selected in the previous step and the Space column filled with "Solution." The user may use the "Change Space" button to change a selected "Solution" entry to "Problem" to indicate that the corresponding noun phrase appears in the statement of the problem to be solved but will not be represented by an Ada type or object in the software solution to the problem. In the third column, the user completes each row of the table by entering an Ada identifier which will be used in the software system being developed. In the final column of the table the user enters an indication of which rows of the table are closely related.

In this example, the student has recognized that "several recipes," "this collection," and "selected recipes" all refer to the same set of recipes, those identified during the baking planning sessions as the ones to be prepared. The student has also decided that "Recipe" and "Number_Of_Cookies" are Ada identifiers which are closely related and should be implemented in the same Ada unit. Also, the student has recognized that the software solution will not have an Ada identifier which directly implements the concept of "cookies."

4.b. Complete the entries in the Object Table.

Object Table

Change Space

Noun	Space	Ada Identifier	Object
recipes	Solution	Recipe	Recipe
a recipe box	Solution	Recipe_Box	Recipe_Box
several recipes	Solution	Selected_Recipes	Planning_Session
the recipe box	Solution	Recipe_Box	Recipe_Box
this collection	Solution	Selected_Recipes	Planning_Session
selected recipes	Solution	Selected_Recipes	Planning_Session
total number	Solution	Number_Of_Cookies	Recipe
cookies	Problem		
the total amount	Solution	Amount_Of_Ingredient	Ingredient
each ingredient	Solution	Ingredient	Ingredient

Step 4a

Step 4c

Prepare Design Report

Print Design

In step 4c, the user is shown a stack of cards, the Object Stack. The user may travel through this stack and add additional information for each object. This information consists of an indication of whether the object will be implemented as a type, an indication of whether the object is an

agent, a server, or an actor, and a comment giving other information about the object. (See Glossary for definitions of agent, server, and actor.)

4.c. Identify the attributes of each object.		
Noun	the total amount	
Space	Solution	Object Ingredient
Identifier	Amount_Of_Ingredient	
Type or Object?	Type	
Agent, Server, or Actor?	Server	
Comment	Associated with each Ingredient is an amount of that ingredient, with dimensions. The system should be able to compute 1/2 cup plus 4 tablespoons giving 3/4 cup.	
Step 4b	Previous	Next
Step 5a	Prepare Design Report	

Steps 5a, 5b, and 5c are similar but deal with verb phrases and their implementation as operations.

5. Identify the operations.

5.a. Select verb phrases in the paragraph.

ENTER recipes into a recipe box. SELECT several recipes from the recipe box. For this collection of selected recipes DETERMINE the total number of cookies TO BE BAKED and the total amount of each ingredient.

Select Verb Phrase

Step 4b **Step 5b** **Prepare Design Report** **Print Design**

5. b. Complete the entries in the Operation Table.

Operation Table

Change Space

Verb	Space	Object	Ada Identifier
Enter	Solution	Recipe_Box	Enter_Recipe
Select	Solution	Planning_Session	Select_Recipe
determine	Solution	Planning_Session	Determine
to be baked	Problem		

Step 5a Step 5c Prepare Design Report Print Design

Step 5c is implemented by the Operation Stack, a stack consisting of one card for each operation in the system under development. The additional information requested for each operation in the Operation Stack at step 5c is an indication of whether the operation will be implemented as a procedure or a function, an indication of whether the operation is a constructor, an iterator, or a selector, conditions under which the operation will not behave properly (for example, attempting to delete a nonexistent component from a data structure), and a comment giving other information. (See Glossary for definitions of constructor, iterator, and selector.)

On the background shared by all cards representing step 5c, the scripts for buttons "Procedure or Function?" and "Constructor, Iterator, or Selector?" give the user the appropriate two or three choices from which to select, helping to insure that only reasonable values are entered into the corresponding fields. The "Comment" and "Error" fields are scrolling

fields to allow the user to enter rather lengthy text which will become Ada comments in the system being developed. The "Verb," "Space," "Object," and "Identifier" fields were automatically filled by the software tool from the Operation Table built in step 5b.

5.c. Identify the attributes of each operation.

Verb Select

Space Solution **Procedure or Function?** Procedure

Object Planning_Session

Identifier Select_Recipe

Constructor, Iterator, or Selector? Constructor

Comment

Errors

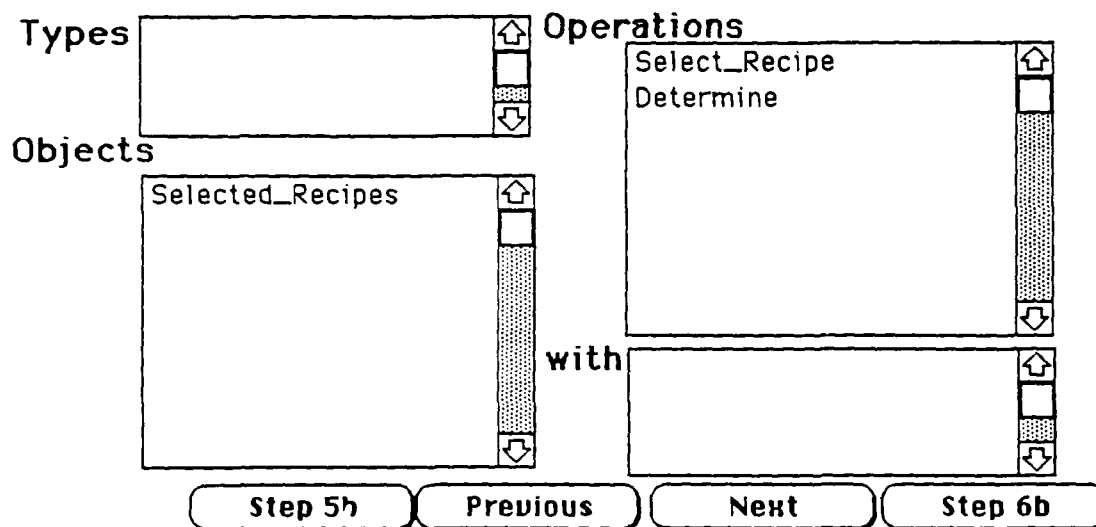
Step 6a allows the user to travel through the cards in another stack, the Unit Stack. This stack is built by the tool by sorting the Object Table and the Operation Table by their Object columns. For each distinct Object column entry in either the Object Table of step 4b or the Operation Table of step 4c a card is created in the Unit Stack. Every object from the Object Table with a particular entry in its Object column will be entered in the Types field or Objects field of the card, and every operation from the Operation Table with that entry in its Operation column will be entered in the Operations field of the card. The "with" field of the card remains empty for the time being. Each card of this stack will be implemented by an Ada package. No activity by the user is requested at this step other than reviewing the stack to determine whether the information presented is reasonable.

In the example, the Object Table entry for object "Selected_Recipes" and the Operation Table entries for operations "Select_Recipe" and "Determine" have all indicated that these items should be implemented as part of one Ada unit, "Planning_Session."

6. Identify the program units.

6.a. Group objects, types, and operations.

Package Planning_Session



In Step 6b the user is shown a "Unit List," with one entry for each card in the Unit Stack. The user selects an entry in the Unit List, clicks the "Choose Unit" button, selects another entry in the Unit List, and clicks the "Choose Needed Unit" button.

In the current example, a user might first choose "Recipe_Box" and then choose "Recipe," indicating that the package implementing "Recipe Box" needs resources provided by the package implementing "Recipe."

After each sequence of selections, the tool updates the "with" field of the appropriate card in the Unit Stack. Whenever the user chooses the "Update Visibility List" button on the current card, the tool examines the cards of the Unit Stack and produces in the "Visibility List" field of the current card a list of all units with an indented list under each one listing the other units to which that unit needs access.

In the example, both "Recipe" and "Ingredient" are indented when they appear under "Recipe_Box" to show that "Recipe_Box" depends on resources provided by "Recipe" and "Ingredient."

6.b. Identify the high-level Ada program units and specify their interfaces.

The screenshot shows a graphical user interface with the following elements:

- Unit List:** A vertical list on the left containing "Planning_Session", "Recipe_Box", "Recipe", and "Ingredient".
- Choose Unit:** A button above the "Unit List".
- Choose Needed Unit:** A button above the "Visibility List".
- Unit Recipe_Box needs unit Ingredient:** A text label indicating a dependency.
- Visibility List:** A larger vertical list on the right containing "Planning_Session", "Recipe_Box", "Recipe", "Ingredient", "Recipe_Box", "Recipe", "Ingredient", "Recipe", and "Ingredient".
- Update Visibility List:** A button above the "Visibility List".
- Step 6a:** A button at the bottom left.
- Step 6c:** A button at the bottom center.
- Update Design Report:** A button at the bottom right.
- Print Report:** A button at the bottom far right.

In step 7a the user is presented with a scrolling field containing Ada package specifications to review, and in step 7b, with a scrolling field containing Ada package bodies to review. At each of these steps a user can click on the "Edit/Print Specifications" button or the "Edit/Print Bodies" to enter Microsoft Word for editing, saving, or printing the contents of either field.

The package specifications produced by the tool at Step 7a are:

```

with Recipe_Box;
with Recipe;
with Ingredient;
package Planning_Session is
  Selected_Recipes: To_Be_Determined;
  -- This object used as a(n) Server
  procedure Select_Recipe;
  -- This operation used as a(n) Constructor
  -- Select_Recipe selects a Recipe from the
  -- Recipe_Box and adds it to Selected_Recipes.
  --Error Conditions:
  -- It is an error if the indicated Recipe is not in the Recipe_Box.
  procedure Determine;
  -- This operation used as a(n) Constructor
  -- Determine examines all Recipes in Selected_Recipes and computes
  -- the sum of the Number_Of_Cookies produced by each and
  -- the amount of each Ingredient needed.
end Planning_Session;

```

```

with Recipe;
with Ingredient;
package Recipe_Box is
  Recipe_Box: To_Be_Determined;
  -- This object used as a(n) Server
  -- Recipes are stored in Recipe_Box.
  procedure Enter_Recipe;
  -- This operation used as a(n) Constructor
  -- Enter_Recipe places a Recipe in the Recipe_Box.
  --Error Conditions:
  -- It is an error to try to Enter a Recipe into a full Recipe_Box.
end Recipe_Box;

```

```

package Recipe is
  type Recipe is private;
  -- Objects of this type are Servers
  -- Each Ingredient consists of the name of the ingredient
  -- and the quantity of that ingredient required for the
  -- given recipe or collection of recipes.
  Number_Of_Cookies: Recipe;
  -- This object used as a(n) Server
private
  type Recipe is To_Be_Determined;
end Recipe;

```

```

package Ingredient is
  type Amount_Of_Ingredient is private;
  -- Objects of this type are Servers
  -- Associated with each Ingredient is an amount of that
  -- ingredient, with dimensions. The system should be able
  -- to compute 1/2 cup plus 4 tablespoons giving 3/4 cup.
  type Ingredient is private;
  -- Objects of this type are Servers
private
  type Amount_Of_Ingredient is To_Be_Determined;
  type Ingredient is To_Be_Determined;
end Ingredient;

```

The Ada package bodies produced by the tool are:

```

package body Planning_Session is
  procedure Select_Recipe is
    separate;
  -- Select_Recipe selects a Recipe from the
  -- Recipe_Box and adds it to Selected_Recipes.
  -- It is an error if the indicated Recipe is not in the
  -- Recipe_Box.
  procedure Determine is
    separate;
  -- Determine examines all Recipes in Selected_Recipes and computes
  -- the sum of the Number_Of_Cookies produced by each and
  -- the amount of each Ingredient needed.
end Planning_Session;

```

```

package body Recipe_Box is
  procedure Enter_Recipe is
    separate;
  -- Constructor places a Recipe in the Recipe_Box.
  -- It is an error to try to Enter a Recipe into a full Recipe_Box.
end Recipe_Box;

```

```

package body Recipe is
end Recipe;

```

```

package body Ingredient is
end Ingredient;

```

The Ada packages produced at this stage by the tool are not adequate for implementing the entire system. The student needs to repeat the process adding new objects and operations to the system.

HyperCard Structure of Our Tool

As described previously, the prototype tool is implemented by four HyperCard stacks, one stack for each of steps 4c (one card per object), 5c (one card per operation), 6a (one card per object or group of related objects, that is, one card per Ada unit), and one stack for the overall system (one card per method step). The cards for steps 4b and 5b (the Object Table and the Operation Table) use multiple scrolling fields and are based heavily on Danny Goodman's (1988b) LaborLog stack.

Some Technical HyperCard Issues

Several technical issues in building the prototype tool were related to the use of HyperCard.

First, there was a need for some text fields to hold arbitrarily long strings. Scrolling fields were an obvious choice, but only the visible portion of these fields is printed when the command "Print Card" is chosen from within HyperCard. To add the ability to print the complete design, most cards were given buttons "Prepare Design Report" and "Print Design." Clicking the "Prepare Design Report" button causes a report based on the fields on this card and the cards for all previous steps to be written into a file. Clicking the "Print Design" button opens the file using Microsoft Word. From within Word the user can then edit, save, and print the file.

Secondly, for ease in examining the cards that refer to major steps in the method, the cards representing individual objects, operations, or units were placed in stacks other than the stack which implements the main path through the method. This design choice has caused the tool to execute more slowly than would otherwise be the case because of continued movement between stacks in some of the button scripts. Speed has been improved somewhat by copying frequently used fields of cards in one stack into local variables of a script so there are fewer switches between stacks.

Status of Project

We currently have a prototype version of the tool for the Berard method of OOD which is being tested in a junior-level Ada course this semester.

From testing the stack, we observe that a production version of the tool should safeguard the internal consistency of the development. For example, if in step 4b a student adds a new entry to the Object Table a note should be entered on the card for step 3 indicating the need for a new sentence with that object. Similarly, when an object or operation (which is not a duplication) is deleted, a note should be entered on the card for step 3.

Also, the ability to add a new sentence to the paragraph in step 3 and then repeat steps 4 and 5 to process any new noun or verb phrases without disturbing those already found should be added. (Currently users are using their knowledge of HyperCard to do this.)

Perhaps more importantly, the tool should provide assistance in adding and modifying parameter lists for subprograms in order to maintain consistency between package specifications and bodies.

Future Work

We hope that a version of the tool described in this paper will eventually be implemented in a high-level language as a collection of structure editors which share a data base. Design of a similar tool for the Booch method of OOD is underway. Future plans include tools to support software development using PAMELA2 and enhanced versions of both OOD tools. The enhanced tools will assist in designing systems which include generic units, packages which export private or limited private types, and tasks (as suggested in Booch (1987b, p. 23)).

Some ideas from Richard Ladden's paper (1989) may also influence the development of the OOD tools.

Notes:

HyperCard, HyperTalk, and Macintosh are trademarks of Apple Computer, Inc.

Microsoft is a registered trademark of Microsoft Corporation.

PAMELA2 is a trademark of George W. Cherry, Thought**Tools, Inc.

References:

Berard, E. V. (1985) An Object Oriented Design Handbook for Ada Software. EVB Software Engineering, Inc.

Booch, Grady (1983). Software Engineering with Ada. Benjamin/Cummings.

Booch, Grady (1987a). Software Engineering with Ada, second edition. Benjamin/Cummings.

Booch, Grady (1987b). Software Components with Ada: Structures, Tools, and Subsystems. Benjamin/Cummings.

Chandhok, R. and Miller, P. (1989) "The Design and Implementation of the Pascal Genie." ACM Computer Science Conference, Louisville, KY, February 1989.

Cherry, George W. (1988). PAMELA2: An Ada-Based, Object-Oriented, 2167A-Compliant Design Method. Thought**Tools, Inc., Reston, Virginia.

Goodman, Danny (1987). The Complete HyperCard Handbook. Bantam Computer Books.

Goodman, Danny (1988a). Danny Goodman's HyperCard Developer's Guide. Bantam Computer Books.

Goodman, Danny (1988b). The HyperCard Handbook 1.2 Upgrade Kit. Bantam Computer Books.

Ladden, Richard M. (1989) "A Survey of Issues to be Considered in the Development of an Object-Oriented Development Methodology for Ada." Ada Letters, volume IX, number 2, March/April, 1989, pages 78-89.

Glossary (from Booch, 1987b):

Actor. An object that suffers no operations but that operates upon other objects.

Agent. An object that serves to perform some operation on the behalf of another object and that in turn can operate upon another object.

Constructor. An operation that alters the state of an object.

Iterator. An operation that permits all parts of an object to be visited.

Selector. An operation that evaluates the current object state.

Server. An object that suffers operations but cannot operate upon other objects.

State. The value and/or object denoted by a name.

AUTHORS INDEX

A

B

Beauman, Roger 87

C

Crawford, Albert L. 161

D

E

Engle, Charles B. 3, 147

F

Feldman, Michael B. 17, 111
Ford, Gary 3

G

H

I

J

K

Kelly, John 121
Klausewitz, Orville E. 191
Koedel, Barbara 171
Korson, Tim 3
Kuhn, Tina 65

L		
	LeGrand, Sue	73
	Levine, Gertrude	31
	Liaw, Morris	11
M		
	Mers, Robert C.	177
	Moran, Melinda	37, 147
	Murphy, Susan	121
N		
O		
P		
	Plain, Russell	99
Q		
R		
	Reedy, Ann	73
S		
	Samuels, Doug	1
T		
U		
	Umphress, David A.	135
V		
	VanScoy, Frances L.	195
W		
	Wall, Mary	171
	Warner, Kathleen	99
	Warner, K.	99
	Wheeler, Orville E.	181
X		

Y

Yodis, Ed	73
Youtzy, Harold	59

Z

NOTES