Transitioning to Ada in an Introductory Course for Non-Majors

A.T. Chamillard and William C. Hobart, Jr.
Department of Computer Science
U.S. Air Force Academy, CO 80840
{ChamillardAT,HobartWC}.DFCS@usafa.af.mil

Abstract
All students attending the U.S. Air Force Academy are required to take an introductory course in computer science. Last year we transitioned to Ada in this course. In this paper we explain our rationale for making this transition and discuss some potential (though not significant) concerns for those considering a similar transition.

1 Introduction

All students attending the U.S. Air Force Academy (called USAFA hereafter) are required to take an introductory course in computer science. While a number of computer science topics are covered in this course, the main focus is on problem solving using computers. Last year, we were faced with some major challenges in restructuring the course. First, to give the students more latitude in managing their time, we were directed to reduce by more than a third the classroom/lab time spent with the students in the course, from 82 hours to 52 hours. Second, while decreasing the total class/lab hours, this course assumed the responsibility for introducing the students to standard office applications including word processors, spreadsheets, databases, and presentation software. Finally, the engineering departments wanted assurances that students successfully completing the course would be equipped to use computers to solve problems in their follow-on courses. Up to this year, we have taught problem-solving using Pascal. Last year, as part of this restructuring, we decided to transition to Ada 95 in this course. In this paper we explain our rationale for making this transition, how this was by far the easiest and least risky of the changes we made to our introductory programming course, and the benefits we have reaped from our transition to Ada 95.

1.1 Course Structure

Computer Science 110 (CompSci 110) is USAFA's introductory course in Computer Science. Because the course is taken by all students in either their freshman or sophomore year, it assumes no prior knowledge about computers. Our overall objective is to enable each student to develop and apply an understanding of the computer's capabilities to serve the Air Force's complex problem-solving needs. To this end, the course has been designed to cover four basic areas of study: problem solving with computers, computing fundamentals, applications of computers in the Air Force, and standard office applications.

Since students need to know how to solve problems before they can solve them using computers, we start by helping the students develop their problem solving skills. They then learn how to use these skills to solve problems using computers and Ada. To help students understand computing fundamentals, we discuss both the hardware components of their computers and other general computing topics (operating systems, networks, and so on). Throughout the course, we introduce the students to standard word processing, spreadsheet, database, and presentation software because they need to use these applications to complete their assignments in their other courses here at USAFA. Finally, we discuss ways in which computers are used to enhance the Air Force's capabilities.

Thus, the course teaches students to take a problem, design a solution to that problem, and implement that solution in Ada. The course also helps students understand more fully how computers work, gives them experience using a variety of standard office applications, and enhances their appreciation of the effective use of computers in the Air Force.

1.2 Rationale for Change

The strongest impetus behind our decision was a desire to prepare future Air Force officers for the environment they will be entering upon graduation from USAFA. Even non-computer science majors are much more likely to be exposed to Ada systems than to systems written in Pascal, so we feel an introduction to Ada is more effective and realistic than an introduction to Pascal. Also, because Ada is the programming language across the computer science curriculum at USAFA, including Ada in the introductory course better prepares our majors for their later courses in computer science.

The next section discusses our transition to Ada 95, and Section 3 presents the results of our transition effort. The final section provides our conclusions based on this effort.

2 Potential Transition Concerns

Despite the reasons cited above, which support a transition to Ada, we had a number of concerns about the logistics and potential pitfalls of the transition effort. These concerns included availability of a suitable environment, faculty inexperience with Ada, scoping the number of language features taught, specifying appropriate programming style, potential difficulties using Ada I/O, and other constraints affecting the course concurrently with the transition effort. Despite these concerns, we decided to go forward with our transition to Ada.

2.1 Providing a Suitable Integrated Development Environment

One of the primary obstacles to transitioning our course to Ada was the lack of a suitable Integrated Development Environment (IDE). We needed an inexpensive, reasonably efficient compiler with a Graphical User Interface (GUI) that was intuitive and easy for inexperienced programmers to use. Mature IDEs meeting these requirements are common for Pascal, but are not as readily available for Ada. The IDE we used during our first semester teaching Ada was provided by the Ada Joint Program Office (alleviating our concern about expense, at least somewhat), but many students found the interface, which was developed with larger projects in mind, difficult to use. We therefore developed our own IDE, AdaGIDE, on top of GNAT, which we used during our spring course offering [1]. We now have AdaGIDE and

1997 ACM 0-89791-981-5/97/0011

GNAT pre-installed on the Pentium II®[1] personal computers purchased by all freshmen for this fall semester at USAFA, and the students should be more than happy with the short compile/build times for their Ada programs.

2.2 Faculty Concerns

A second concern was faculty inexperience with Ada. Though of course all faculty members had heard of the language, only a small number had actually programmed in Ada. It was therefore not clear to us that we would be able to develop sufficient faculty mastery of Ada within the department[2] to teach Ada in the course. It turns out, however, that Ada is sufficiently similar to Pascal and other imperative languages that our faculty was able to quickly develop a mastery of the language, at least for the subset we deemed appropriate for our course. This is confirmed by student critiques ranking the computer science faculty expertise in the top 25% of all core courses at USAFA.

2.3 Scoping the Language Features

This brings us to our third concern -- scoping the number of language features taught. Ada is a feature-rich language, and we were concerned that covering too many of these features would overwhelm the students. This is particularly important because our goal in the course is to develop problem solvers, with Ada as the tool used to implement those solutions; we are not attempting to create expert Ada programmers. After careful consideration of the types of problems solved in the course, we selected the language features listed in Figure 1. This subset gives us sufficient variation in data types to allow solving most basic problems, gives us all three control structures (sequential, selection, iteration), gives the students the tools to develop modular solutions, provides sufficient IO capability to solve the problems we need to solve in the course, and lets the students use the standard packages provided with Ada. We note that this "Pascal subset" is very similar to the Ada features taught by Weiderman and Coulter in their validation of teaching Ada as an introductory programming language [2]. We were particularly concerned about difficulties the students might have with IO, since our problems are typically IO-intensive (rather than computation-intensive) and Ada IO seems somewhat more complicated than Pascal IO.

Data Types:
    Integer, Float, Character, String, Boolean,
    One-dimensional Arrays
Statements:
    Assignment
Control Flow:
    If-Elsif-Else, For Loop, While Loop
Modularity:
    Packages, Procedures
Input/Output:
    Put, New_Line, Get, Skip_Line, File IO (Open,
    Create, Close)
Libraries:
    With, Use

Figure 1. Ada Features Used in the Course

[1]Pentium II is a registered trademark of Intel Corporation

[2]Because we typically teach over 500 students in this course every semester, almost every member of the department teaches at least one section of the course (most teach 2 or 3 sections).

We teach the use of both packages and procedures to implement modularity, but we do not require the students to write packages. There are two main reasons for this. First, we use Ada as a way to teach problem solving and build the students' confidence in using computer to solve problems. We have found that our students are able to grasp the concept of breaking a problem into sub-problems and then implementing those sub-problems with procedures. We can therefore develop essential problem-solving skills without requiring packages in the student solutions.

In addition, we do not require the students to write packages because the problem solutions they implement are too small to justify package development. For example, the largest problem they need to solve in this course is the final comprehensive programming project; an excerpt from the assignment is provided in Figure 2.

Statement of Requirements for an Assignment Tracking System

You've been tasked to develop a system for tracking information on up to 100 class assignments. Specifically, your system shall allow the following 5 information elements to be retrieved, displayed, or stored for each assignment:

Class Name -- a 15 character string (i.e., "Comp Sci 110 ")
Assignment Name -- a 10 character string
Point Value -- an integer
Class Day -- a character, only M or T
Lesson Due -- an integer between 1 and 42

Your system shall start by displaying a short overall description of its purpose followed by a menu of choices for the user that includes the following:

(R)ead a file as input to initialize the database with assignment information
(W)rite a file as output to store the current assignment information
(D)isplay all assignment information in the database on the screen
(A)dd a new assignment to the tracking system database
(F)ind assignments in the database and display them, given the class name
(Q)uit the program

The system shall then prompt the user for a one-character choice. Once the user has entered his or her choice, the program shall accomplish the desired action and then reprompt the user for another choice. If the user enters an invalid choice, the system shall display an error message and reprompt the user for a valid choice until one is entered. The system shall accept both lowercase and uppercase inputs, i.e., either 'A' or 'a' to add a new assignment to the tracking system database.

Figure 2. Excerpt from Final Comprehensive Programming Project

We believe a "correct" implementation of a solution to this problem includes approximately 6 procedures and the use of parallel arrays. Using a package to hide the implementation of the database with parallel arrays could result in an improved design, but it is difficult to rationalize this approach without

presenting an alternative implementation, such as using an array of records or some other language feature beyond the scope of those taught in the course.

Although we do not require the students in this course to write packages, we do require that they use them. For example, students use the standard Input/Output packages for text, floats, and integers. We also have the students implement a small text-based case study in small groups. We then show the students how we can convert this to a graphics-based implementation by replacing the Withs and Uses on the standard text packages to Withs and Uses on a graphics package one of the course instructors implemented. We therefore reinforce the value of packages as a way to implement modularity without requiring that the students actually implement their own packages. While we recognize that the ability to conceptualize abstract data types and implement them using packages is essential knowledge for our computer science majors, we do not have the time to teach this to the non-majors in this introductory course. Instead, we reserve this topic for our CS1 and CS2 courses [3].

An understanding and ability to use the language features discussed above will let students successfully complete the course, but we do provide extra credit opportunities for those students who would like to explore the language further. The extra credit opportunities are not "open-ended"; rather, we guide them toward other interesting topics and offer them a set number of points for learning to use other constructs in particular assignments. These additional constructs include records, exception handlers, and so on. We strongly suggest that students complete the "base assignment" first so that they get full credit for the assignment before trying the extra credit portions (which they turn in as separate programs). If they use this approach, students can avoid getting a lower grade if their (mis-)use of the extra credit constructs adversely affects the functionality of their programs.

2.4 Specifying Programming Standards

After scoping the language features taught, we still needed to specify programming standards. Capitalization and indentation can be automatically accomplished by the IDE we developed, but other issues were not solved so easily. For example, should we let the students take advantage of Use clauses? All of the faculty agreed that Use clauses can cause difficulty understanding larger systems, particularly those with overloaded functions and procedures, but these concerns are not applicable to the programs students write in this course. In the first semester we did not allow Use clauses, while in the second semester we did; results are reported in Section 3.3. We also decided that procedure calls should be made using named association for the parameters. With Pascal we often found errors in number, order, or type in procedure calls, and using named association at least removes the order problem.

2.5 Teaching Ada in Half the Time

Our final concern was not about Ada per se; our course was going through a restructuring at the same time we were transitioning to Ada, and we were concerned that this would adversely affect our ability to include Ada in the course. Specifically, the contact time with students was being reduced from a total of 40 hours to a total of 20 hours for the programming portion of the course. We believed that cutting the programming contact time in half might reduce our ability to teach, and the students' ability to learn, Ada in sufficient detail to implement the problem solutions required.

3 Transition Results

In general, the students in this course do not appear to have any more difficulty learning Ada than they did learning Pascal, though there is of course a learning curve in terms of using both the IDE and the language itself. To evaluate the students' programming skills, we compared the students' programming performance on the Spring 1996 final exam (using Pascal) with that of the students taking the final exam in Spring 1997 (using Ada). Results of this comparison are provided in Table 1.

Although we note that the exams contained different problems and had a different format (closed versus open book), the mean scores on the two exams are quite close. Thus, not only is Ada as easy, if not easier, to learn for those majoring in computer science as summarized by Feldman [4], but it also seems to be easily grasped by those who may be much less technically oriented.

3.1 Providing a Suitable Integrated Development Environment

Our IDE seems to be reasonably easy to use for the inexperienced programmers in the class (though it contains more advanced features as well), but they still have difficulty interpreting compiler error messages. We believe this is due in part to their inexperience with the language, but also is indicative of their resistance to actually reading the messages carefully to see what they say. We try to use compiler errors to develop student analytical skills by having them read the errors (sometimes aloud), then having them rephrase these errors in their own words and investigate the source of the problem.

We also plan to add context-sensitive help to the IDE, which should aid the student debugging process. We currently provide a help facility that gives descriptions, syntax, and examples of the constructs used in the course, as well as an on-line Language Reference Manual. We plan to augment these reference

Comparison	Spring 1996 Pascal Final Exam	Spring 1997 Ada Final Exam
Topics Tested	Specifying algorithms, Declaring constants and variables, Using assignment statements, Declaring procedures, Using multiple-alternative-IF, Declaring and indexing arrays, Modifying design for changed problem	Using WITH and USE statements, Declaring constants and variables, Using assignment statements, Declaring procedures, Evaluating Boolean expressions, Evaluating and using multiple-alternative-IF, Evaluating nested FOR loops, Evaluating and using WHILE loops, Declaring arrays, Evaluating and using indexing of arrays
Format of Exam	Closed book	Open book for programming portion (Ada 95 text only)
Students Taking Exam	557	462 (top 5% excused from exam)
Mean Score on Programming Portion of Final Exam	73.2% (estimated from sample of 4 out of 27 sections)	77.1%

Table 1. Comparison of Student Performance on Pascal and Ada Final Exams

materials with context sensitive help that, given a particular syntactic error in the program, will automatically raise a window with help on the construct causing the error.

We recognize that another way to help the students interpret error messages would be to more quickly build their Ada expertise, perhaps using either a new or existing on-line tutorial. We are not currently pursuing this option, however, because our main goal is to develop problem solvers, not Ada programmers. In addition, surveys indicate that the course currently consumes the appropriate amount of student time given that it is a 3 semester hour course. Because most students taking the course are carrying loads of between 16 and 22 semester hours, we do not believe that adding tutorial time at the expense of their other activities in this course or their other courses is a viable alternative.

3.2 Scoping the Language Features

It also seems that we have properly scoped the subset of Ada features taught. Most students are not overwhelmed with the features we cover, yet these features provide sufficient capabilities for implementation of all the problem solutions required in the course. The extra credit opportunities let students get credit for exploring the language further, and the constructs we include in the extra credit opportunities seem to be properly scoped.

3.3 Specifying Programming Standards

Using named association in parameter calls seems to be working well, both when we teach the concept of parameter passing and when the students actually implement the code to use procedures. Named association eliminates the out-of-order problem in parameter calls, one of the most difficult problems for our Pascal students to debug in previous years.

When we disallowed Use clauses in the first semester, student tended to believe that much of Ada was a "haze", particularly given the number of times they needed to prepend "Ada.Text_IO." to their procedure calls. This propagated the myth that Ada is difficult to use and also affected student morale, so we decided to allow Use clauses during the second semester. This has worked well - students don't complain so much about the typing and seem to have a more positive attitude about the language, while we have not increased instructor difficulty understanding what student programs are doing.

3.4 Teaching Ada in Half the Time

One of the primary reasons we were able to successfully teach Ada (or any programming language) despite our time constraints was the availability of course instructors for office hours. A typical instructor in this course offers 3 to 5 office hours per day, with extended office hours provided during the final comprehensive programming project. We believe that 15 to 25 hours per week for one-on-one instruction is somewhat atypical in the academic community.

Another technique we used to help resolve the time problem was to allow and encourage collaboration during the first part of the course. This let the students learn from each other and reap the other benefits associated with collaborative learning.

Through use of the extensive instructor office hours and collaboration with their classmates, students in this course were able to master the essential programming constructs and were, for the most part, successful in individually completing the final comprehensive programming project.

4 Conclusions

The transition from Pascal to Ada is both easy to make and highly beneficial to the students. Faculty members who are familiar with other imperative languages can master the subset of Ada necessary for an introductory course very quickly. Efficient and easy-to-use IDEs for Ada are becoming increasingly available, and both AdaGIDE and the GNAT compiler are available at no cost. Ada very effectively supports all essential constructs required to teach problem-solving, and there is a direct correspondence between Pascal and Ada for these constructs. Finally, where the two languages do differ, Ada provides facilities (like named association) to make problem-solving constructs (such as procedures) easier to understand. In addition, Ada provides a high level of "compiler safety" (e.g., type checking) that is particularly important for the inexperienced programmers in this course. We believe that we have effectively integrated Ada into our introductory course and we will continue to use Ada both in this course and across the computer science curriculum.

References

[1] Martin C. Carlisle and A.T. Chamillard, AdaGIDE : A Friendly Introductory Programming Environment for a Freshman Computer Science Course, In *Proceedings of the Eleventh Annual ASEET Symposium*, Monmouth, New Jersey, June 1997.

[2] Nelson Weiderman and Virginia Weiss Coulter, Using Ada as an Introductory Programming Language, *Journal of Pascal, Ada, and Modula-2*, Vol. 6, No. 1, pp. 10-34, 1987.

[3] John A. Hamilton and David A. Cook, Ada Training and Education in the US Army and US Air Force, In *Proceedings of TRI-Ada '96*, Philadelphia, Pennsylvania, December 1996.

[4] Michael B. Feldman, Ada Experience in the Undergraduate Curriculum, *Communications of the ACM*, Vol. 35, No. 11, pp. 53-67, 1992.