Presented at the 2002 Rocky Mountain Conference of the Consortium for Computing in Small Colleges

Evolution of an Introductory Computer Science Course: The Long Haul

A.T. Chamillard and Laurence D. Merkle
U.S. Air Force Academy

A.T. Chamillard
P.O. Box 384
Fort Belvoir, VA 22060
703-428-0528
achamillard@hq.dcma.mil

Laurence D. Merkle
HQ USAFA/DFCS
2354 Fairchild Drive
USAFA, CO 80840
Larry.Merkle@usafa.af.mil

Abstract

University requirements for the material covered in introductory computer science courses have evolved over the years, and those courses must therefore evolve as well. In this paper, we discuss the 7-year evolution of such a course at the U.S. Air Force Academy. In 1995, the main thrust of the course was to develop students' programming skills to support later programming activities, even for those students not majoring in computer science. Although some general survey topics were covered, programming skill development was the main goal of the course. Since that time, the course has evolved significantly into a course that covers general computer science and Information Technology (IT) topics in greater depth and breadth, with a continuing but greatly reduced programming component. During that 7-year period, we changed programming languages for the course, significantly changed the way in which we evaluated programming ability, incorporated graphics into the course, conducted an extensive rework of the course content, and made numerous smaller changes as well. In this paper, we discuss the technical and political issues associated with the evolution of the course. Although this work is presented in the context of our course, such evolution is clearly applicable to other introductory courses as well.

1. Introduction

All students at the U.S. Air Force Academy take an introductory course in computer science in either their freshman or sophomore year. The course covers both programming topics and non-programming topics, such as computer hardware organization, operating systems, networking, databases, the World Wide Web, and so on.

Because the field of computer science evolves, and because University requirements for this kind of course also change over time, such courses must evolve as well. Topics need to be regularly updated to remain current and relevant, various aspects of the presentation sequence and content need to be modified repeatedly, and, on occasion, the entire course may need to be redesigned to more effectively cover the changing learning objectives of the course.

The following section discusses our language change from Pascal to Ada. Section 3 presents the ways in which we have evolved our evaluation of student programming ability, and Section 4 describes our incorporation of graphics in the course. Our recent ground-up redesign of the course is addressed in Section 5; the final section presents our conclusions.

2. Programming Language Change

One of the challenges regularly faced by curriculum developers for introductory courses is the transition from one programming language to another. The factors driving such transitions include: a desire on the part of the computer science faculty to use a “current” language, perhaps to improve the marketability of their graduates; changes required to support later classes in the computer science curriculum; requests from other departments to use a different language to support their courses; and political pressures from other departments and administrators to use the “latest” language, both to demonstrate currency and to support recruitment and retention.

Many departments are currently deciding whether or not to move from C++ to Java, from Java to C#, or even from Pascal to C++ or Java [5]. Even TCL/TK has been proposed as a suitable language for the first computer science course [7]! In 1996, the Air Force Academy decided to transition the language used in the introductory computer science course from Pascal to Ada [2]. This transition was motivated in large part by the fact that Air Force Academy graduates are much more likely to encounter Ada systems rather than Pascal systems in their careers, though we also point out that other departments (most notably, the Department of Astronautics) decided to use Ada in their courses as well. Because there was evidence in the literature that Ada was not difficult to learn for those majoring in computer science [6], we believed that we could effectively integrate Ada into our introductory course.

There are numerous issues to be considered when transitioning to a new programming language. Selecting the new language is, of course, a major decision point. Choosing an appropriate development environment is another critical decision, especially given the programming inexperience of many of the students in the course. Other issues include ensuring faculty members are prepared to teach the new language, scoping the language features covered to be of reasonable difficulty for the students, and selecting the appropriate programming standards to be used in faculty and student programs.

A primary concern that educators face when changing languages is how the change will affect student learning and performance in the course. To examine these effects from our 1996 transition to Ada, we compared student performance during the Spring 1996 (using Pascal) and Spring 1997 (using Ada) semesters on the programming assignments, on the end-of-semester programming exercises (PEXes), and on the final examinations. We chose these assessments as those that were most directly related to evaluation of student programming ability.

	Spring 1996 (Pascal)	**Spring 1997 (Ada)**
Programming Assignments	82.3 %	90.3 %
PEX	81.9 %	89.3 %
Final Exam	77.9 %	80.4 %

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

We note that the final exams contained different problems and had a different format (closed versus open book), but seemed to be of comparable difficulty, at least from the perspective of the instructors for the course. The mean percentages of the assessment scores for these semesters are provided in Table 1.

Informally, the results in Table 1 indicate that the students performed better on the programming assessments when we used Ada in the course. More formally, we can use a standard t-test to compare the means of the distributions for each assessment from the Spring 1996 and Spring 1997 semesters. The t-test uses the means and standard deviations for the two distributions to try to reject a null hypothesis that the two distributions could have been drawn from populations with equal means. For all three of the assessments under consideration, the t-test yielded results indicating that the means were in fact higher, with statistical significance, for the semester in which Ada was used as the programming language.

Based on these results and other anecdotal observations, we view our transition from Pascal to Ada in this course as a success. The transition from Pascal to Ada was both easy to make and highly beneficial to the students. Faculty members who were familiar with other imperative languages easily mastered the subset of Ada necessary for an introductory course; we expect the same would be true for a transition from one object-oriented language to another. Finally, and most importantly, students performed better on the programming portion of the final examination when Ada was used in the course rather than Pascal.

In response to some of the factors enumerated above, we are now considering another language change. We are evaluating a number of different possible languages, including C, C++, C#, and Java, though we have not yet finalized plans for such a transition.

3. Evaluating Programming Ability

Another major evolutionary change we made involved the evaluation of student programming ability in the course. All 7 years of the course have included assessment of this programming ability through the use of programming assignments (with varying levels of collaboration allowed), test questions, and final examination questions. Prior to the 1997 Academic Year, the course concluded with a large Programming Exercise (PEX) that students were required to complete without collaborating with their fellow students. Our intent with this "individual effort" programming assignment was to evaluate each student's individual programming ability.

An example of the statement of requirements for a PEX is provided in Figure 1. The PEX consisted of 3 or 4 turn-ins designed to help the students develop their solutions iteratively. Each turn-in was comprised of the student's design (in the form of a detailed algorithm) and code for a pre-determined portion of the assignment requirements. Students generally worked on the PEX for the last month or so of the course; because this assignment was worth 15% of the course grade, it was a source of great stress for the students and faculty.

The PEX was stressful for the faculty because, although students were prohibited from working with their fellow students on this assignment, they were encouraged to work with their professor or other professors in the department when they encountered problems. The students embraced this approach wholeheartedly, and professors were known to provide 6 or 7 hours of help to students in a single day, especially as the turn-in due dates approached. It seemed clear,

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 1. Excerpt from Programming Exercise

at least anecdotally, that we were not evaluating each student's individual programming ability given this level of professor participation in the PEX.

In fact, we also had statistical support for the claim that the PEX was not effective for this purpose. In the Spring 1996 semester, the average student percentage on the collaborative programming assignments was 82.3% and the average student percentage on the individual-effort PEX was 81.9%; for the Spring 1997 semester, these percentages were 90.3% and 89.3%. Application of a standard t-test indicated that these percentages were not different with statistical significance, indicating that we were essentially "measuring the same thing," which in turn indicated to us that the PEX was not assessing individual student programming ability. Given both the anecdotal and statistical evidence, we decided to pursue an alternative to the PEX

In the 1997 Academic Year (and in the following years), the PEX was replaced by a combination of a group case study and lab practica [3]. For the case study, we divide the students into teams (of 2 to 4 students) and have them program a portion of a large program. Historically, this program has been a game of some sort; in the recent past, students have implemented portions of Connect-4, Battleship, Othello, Mancala, and billiards. On the last day of the class, we play the student programs against each other in a tournament, and allocate extra credit points based on group placings in the tournament. While it's not clear that the students gain additional programming skill while completing the case study, it does serve as a very motivational experience for the students.

Students complete two lab practica over the course of the semester. A lab practicum is an in-class lab that the students are required to complete within a set period of time. Students must develop and test a complete program solving a problem that they are presented with at the beginning of the time period. They are allowed to use a handout containing syntax for all the programming constructs covered in the course, a sheet listing common programming errors (and their solutions), and the course web site. They are not allowed to use any other materials, and the instructors will only answer questions about the problem (rather than helping students correct syntax errors, for example). In essence, these practica serve as programming exams that test the students' individual programming skill, an assessment we did not feel we were accomplishing with the PEX.

The problem statement from one version of the second practicum is provided in Figure 2. Since it is sometimes difficult to precisely describe required graphical output for a program, the practicum handout also contains an example of the required output.

Develop an Ada program that will do the following:

1. Display a *short* introductory message in the text window (do not require the user to press return!).
2. Open a graphics window that is 500 pixels wide and 300 pixels high.
3. Get ten left mouse button clicks from the user. After each mouse click your program should:
   a. Write in the text window the number of mouse clicks that have been recorded so far
   b. Store the x and y coordinates from each mouse click in an X array and a Y array
   c. After the first mouse click, draw a line (any color) from the previous mouse click to the current mouse click
4. Find the maximum value in the X array
5. Find the maximum value in the Y array
6. Create a file called “A:\Practicum_2a.dat” and write the information described in the format shown below. Note: do NOT write the string “Max X” in the file; write the **value** that is the maximum X value
7. Wait 5.0 seconds and close the graphics window.

Figure 2. Problem Statement from Practicum 2

Because the prime motivation for our evolution to this new approach was to provide more effective assessment of individual programming capabilities, we compared the average students percentages on the in-class practica with the average student percentages on the collaborative programming assignments. Recall that the PEX did not give us a different mean from the programming assignments; we view a statistically significant difference between the practica percentage and the programming assignment percentage as a prerequisite for claiming that the practica give us a more effective method for assessment. A comparison of these percentages for the Spring 1998, Spring 1999, and Spring 2000 semesters is provided in Table 2.

	Spring 1998	**Spring 1999**	**Spring 2000**
Programming Assignments	96.5 %	92.7 %	93.8 %
Lab Practica	78.1 %	74.8 %	77.6 %

Table 2. Comparison of Student Performance on Collaborative Assignments and Lab Practica

Not surprisingly, application of a standard t-test indicates that these percentages are in fact different with statistical significance. We therefore believe that we have identified a means for assessing individual student programming ability that is more effective than the PEX, and we continued to use the practica for this assessment until the course re-work discussed in Section 5.

4. Using Graphics

Prior to the Fall 1999 semester, all the evaluation techniques except the case study were text-oriented. For the case study, we provided all the graphics required; students simply wrote input, processing, and output subprograms without being concerned with how the graphics worked. Starting in the Fall 1999 semester, we incorporated graphics into all our student programming assignments and other assessment techniques [4].

One of the primary reasons for doing this was to provide motivation for the students as they tried to learn new, sometimes difficult, concepts. The students enjoyed the graphical capabilities of the case study code we provided to them, so it seemed reasonable to let them try some of the graphics input and output on their own. Others have also recognized the benefits of using graphics in early computer science courses. Roberts points out that students are much more enthusiastic about writing programs containing graphic functions [8], and graphics have been incorporated in general education computer science courses [9] as well as more traditional CS1 courses [1].

We incorporated graphics in all 6 of the programming assignments in the course, in the lab practica, and in the case study. Although we believe that we incorporated graphics into these assessments in a logical manner, there are clearly some risks associated with incorporating graphics into this course. For example, the students could become so engrossed with the graphics that they overlook important programming concepts. Similarly, students could become entangled in the syntax required to use graphics routines to the detriment of more general topics.

One way to determine whether or not the risks mentioned above are having an effect on student performance is to examine student grades on the programming assignments to determine whether the students do better on text-oriented or graphics-oriented assignments covering similar key topics. Because the order of topic presentation changed significantly in the course concurrently with our inclusion of graphics, we only compared student grades on Assignment 1 from the Fall 1999 and Fall 1998 semesters. In both of these semesters, the key topics for this assignment were variable declarations and use and data input and output. The only significant difference between the Fall 1998 and Fall 1999 Assignment 1 is the addition of graphics to the Fall 1999 lab; a comparison between the two is therefore both reasonable and enlightening.

We first examined the means for this assignment. The mean score on Assignment 1 in the Fall 1998 semester was 75.7% with a standard deviation of 23.2%, while the mean in the Fall 1999 semester was 87.8 % with a standard deviation of 19.7%. While these means are clearly different, we would have liked to run an independent samples t-test to quantify the significance of the difference. However, Kolmogorov-Smirnov tests and graphs of the grade distributions indicate that the distributions are not normal; therefore, a t-test would not be an appropriate comparison statistic for the means. Informally, however, it is clear that the students performed much better on the graphics-oriented lab (Fall 99) than on the text-oriented lab (Fall 98).

While the grade comparison presented above shows that incorporating graphics has improved student performance on this assignment, we would also like to quantify the motivational benefits we are reaping by including graphics. To do this, we suggest that, if graphics-oriented labs are more motivational than text-oriented labs, fewer students will "give up" as they try to complete the assignment.

We therefore counted the number of students that received a grade of less than 33.3% in each semester. We recognize that this may also capture students who did not give up and were simply unable to master the material, but we suspect this was a small percentage of the students who received such low grades on the assignment. In the Fall 1998 semester, 39 of 528 students (7.39%) scored lower than a 33.3% on the assignment. In contrast, in the Fall 1999 semester, only 14 of 467 students (3.00 %) scored lower than 33.3%. While we recognize that this metric is at best an indirect measure of motivation, we do believe that the difference between the text-oriented and graphics-oriented semesters provides further support for our conclusion that incorporating graphics is beneficial.

5. Recent Rework

In the Fall 2001 semester, we implemented a ground-up redesign of the course. The primary purpose of the redesign was to make the course more relevant to students who do not declare computing-related majors. In previous years, the course focused heavily on programming topics and addressed other topics only at the knowledge and comprehension learning levels. This was appropriate for students who would eventually choose a computing-related major, but not for the vast majority of students enrolled in the course. We met regularly as a department over a period of several months to determine what material to include in the redesigned course. Eventually, we decided to increase the emphasis on a number of non-programming topics, including algorithms, hardware, operating systems, networking, World Wide Web, security, multimedia, databases, modeling, and simulation. We developed application-level learning objectives for each of these topics.

At the same time, we incorporated a number of other changes to improve the effectiveness of the course for all students:

- We organized the course into “blocks” of lessons. We had previously only used this structure for programming topics.

- We added web-based pre-assessment quizzes covering the reading for each block at the knowledge and comprehension learning levels.
- In the process of redeveloping the lesson plans, we replaced lectures with active and collaborative teaching techniques wherever we could. This was particularly effective in combination with the pre-assessment quizzes.
- Each incoming class buys standard computers (the configuration is selected and standardized by the Air Force Academy each year); the Class of 2005 was the first class to receive notebook computers. We required students to bring their computers to class, and we incorporated their use in a majority of the lesson plans.
- We use Ada as the high-level programming language for the course because it is very easy to learn. Nonetheless, it has a few syntactic structures that are unnecessarily complicated for an introductory course (e.g. declaration of an array variable requires a type definition). We developed a "CS110" package that simplifies the syntax (e.g. by providing standard array types) to let us focus on the principles behind the programming constructs.
- There is great variety in the backgrounds of incoming students. Because of this, a few students in each section were not challenged by the course in the past. We therefore instituted an Honors version to challenge these more advanced students. The Honors version covers the same topics as the regular version in greater depth, as well as covering additional programming topics.

These changes produced outstanding results. Most importantly, instructors felt that their students learned more than in previous semesters. Objectively, the students scored better on exams, supporting this belief.

The pre-assessment quizzes had the intended effect: the students were better prepared for class, so we could use class time to focus on the more difficult application level learning objectives. The drawback was that our first implementation of the quizzes was based on a free plug-in for Microsoft FrontPage, which turned out to somewhat inflexible and unreliable. We have since developed our own implementation that seems to be more effective for us.

Finally, despite the course's bad reputation, it received outstanding student critiques. Specifically, students gave it one of the lowest ratings at the Air Force Academy for the statement "Prior to taking this class, I was interested in the content of this course," but at the end of the semester, they rated it the highest out of the 11 core course in the Basic Sciences Division and the Engineering Division in 9 of 36 categories:

- Intellectual challenge and encouragement of independent thought
- Evaluative and grading techniques (tests, papers, projects, etc.)
- The course as a whole
- Amount I learned in the course
- This course improved my ability to deal with problems that don't have an approved solution
- My motivation to learn has increased because of taking this course
- There are a number of things in this subject I'd like to learn more about
- My instructor designed activities that made me think
- I believe all the information contained in this critique is anonymous

6. Conclusion

In order to reasonably reflect the modern capabilities of computing technology and its uses in society, the content of introductory courses in computer science must continually evolve. Occasionally, the shift in content is sufficient to demand corresponding changes in course organization and student activities.

In the mid 1990's, the Air Force Academy's introductory course was focused on programming. Several changes were put in place to help students learn essential programming concepts, and to assess that learning. One change aimed directly at improving learning was the introduction of a new programming language. Other changes that indirectly improved learning by increasing student motivation included the incorporation of graphics and the addition of a group project involving a game. Changes aimed at improving assessment of student learning included the elimination of individual effort final projects and the addition of programming practica.

More recently, the course has been redesigned from the ground up, primarily to improve its relevance to students who do not choose majors related to computing. These students are much less likely to need programming skills, but still need to be able to effectively leverage computing technologies. The redesigned course spends less time on programming, but addresses non-programming topics in non-trivial depth. Those topics include algorithms, hardware, operating systems, networking, World Wide Web, security, multimedia, databases, modeling, and simulation

Bibliography

[1] Astrachan, O., and Roger, S.H. Animation, Visualization, and Interaction in CS 1 Assignments, In *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, Atlanta, Georgia, March 1998, pp. 317-321.

[2] Chamillard, A.T. and Hobart, William C. Transitioning to Ada in an Introductory Course for Non-Majors. In *Proceedings of TRI-Ada '97*, St Louis, Missouri, November 1997, pp. 37-40.

[3] Chamillard, A.T. and Joiner, Jay K. Using Lab Practica to Evaluate Programming Ability. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, February 2001, pp. 159-163.

[4] Chamillard, A.T., Moore, Jason A., and Gibson, David S. Using Graphics in an Introductory Computer Science Course. *JCSE Online*, February 2002.

[5] Dingle, Adair and Zander, Carol. Assessing the Ripple Effect of CS1 Language Choice, *The Journal of Computing in Small Colleges*, Vol. 16, No. 2, pp. 85-93, 2001.

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

[7] Isaacson, Peter C. An Introduction to TCL/TK: The Best Language for Introduction to Computer Science Courses, *The Journal of Computing in Small Colleges*, Vol. 16, No. 2, pp. 115-117, 2001.

[8] Roberts, E.S. A C-based graphics library for CS1, *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, 1995.

[9] Stegink, G., Pater, J., and Vroon, D. Computer Science and General Education: Java, Graphics, and the Web, In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, New Orleans, Louisiana, March 1999, pp. 146-149.