A Comparison of Ada and Pascal in an Introductory Computer Science Course

Major Jeanne L. Murtagh, USAF
Software Professional Development Program
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-7765
jmurtagh@afit.af.mil

Lt. Col. John A. Hamilton, Jr., US Army
D/Electrical Engineering & Computer Science
United States Military Academy
West Point, NY 10996-1787
dj7560@exmail.usma.edu

1. ABSTRACT

Pascal's long and successful run at West Point illustrated Winston Churchill's dictum that, "The absence of alternatives clears the mind wonderfully." However, by the mid-90s, the United States Military Academy was finding that Pascal, long the educational workhorse, was no longer the clear choice for serious computer science education. The decision was taken to evaluate alternatives to Pascal. The criteria selected for programming language evaluation were student learning outcomes based on course goals and objectives. The controlled educational environment of a service academy provides an excellent opportunity to conduct research using control groups and experimental groups. We conducted a side by side comparison of the use of Pascal, which was specifically designed for instructional purposes, and Ada in an introductory course. The experiment demonstrated that students were much more successful in Ada than in Pascal, and led to the revision of the Academy core curriculum to use Ada 95 in our introductory computer science class. This paper discusses the details of our comparison, citing specific examples to illustrate a rational basis for evaluating programming language features against course objectives.

Introduction

Every cadet who attends West Point is required to complete CS 105, Introduction to Computers and Programming. CS 105 is a CS 0 course by ACM standards. The course is one semester long, with thirty-four lessons of fifty-five minutes and six two-hour labs. CS 105 consists of three parts: ten lessons on computer concepts, twenty-three lessons on problem-solving through programming, and seven lessons on the use of office automation applications. This paper is based on the results from the programming block of the course.

2. SETTING THE STUDY PARAMETERS

The primary issues in any curricular decision regarding CS 105 are the goals and objectives of the course:

CS105 Course Goals

- Become familiar with the fundamental concepts of computer science
- Develop proficiency in an engineering problem solving and design methodology
- Understand the importance of advanced information technologies

CS105 Course Objectives

- Use computers and application software as tools to solve problems
- Analyze, design, build and test operational solutions
- Apply the building blocks of Sequence, Selection, and Iteration as the foundation of algorithmic processes
- Learn to exploit the educational and professional resources available on the Internet and World Wide Web
- Develop a framework for considering the ethical implications of advanced information technology

While it is clear that a high order programming language is needed to achieve several of these goals and objectives, there are many languages that have the potential to be successful. The Reid Report, compiled and updated by Dr. R. J. Reid of Michigan State University[10], indicates that the primary programming languages in academic use today are, in no particular order, Ada, C/C++, Pascal, Modula and Scheme. We considered a number of issues as we decided which language(s) to evaluate as a replacement for Pascal, and determined that many of those languages could be eliminated from further consideration. The unsuitability of C and C++ for educational purposes is well documented. The confusing syntax is detailed in [9], and the lack of standardization of C++ is discussed in [1]. Even with the recent successful balloting of the C++ standard, popular

C++ compilers come with many proprietary features. There was little perceived gain in moving from one flavor of Pascal to one flavor of Modula. Scheme is not used anywhere at West Point and there was little or no faculty expertise in Scheme, whatever its merits. At the time of the study, Java was not well-known and the procurement of a large number of Java compilers was considered imprudent. Ada seemed to be the best choice for further consideration.

However, there were some concerns about the complexity of the language and its development environments. Ada is renowned as an excellent language for very large software development projects. It is a powerful language with many features which support advanced software engineering concepts. And Ada is still without peer in the high reliability domains [7]. Would the language, as some have claimed, be too complicated for novices for whom readability and simple syntax are more important pedagogical requirements [11]? We were also concerned about availability of a suitable development environment. The compiler environment used in CS 105 must have strong ease-of-use features and run efficiently on the desktop computers. PC-targeted Ada compilation environments did not meet these requirements a few years ago. However, the development of low-cost, high-quality Ada compilers combined with more powerful desktop PCs warranted consideration of Ada for our CS 0 course. Hence, our hypothesis was born: Ada would be a good replacement for Pascal in our CS 105 class.

It was important for us to have sound evidence that this hypothesis was true. Although Ada has been used successfully in many computer science programs, so have many other programming languages. Many of the success stories are anecdotal in nature [8]. The rigorously structured USMA curriculum and the large number of students (more than 30 sections of CS 105, with 18 students per section, each semester) required more than anecdotal evidence or professorial whim before changing languages, particularly since Pascal had been designed specifically for teaching.

3. CONDUCT OF THE EXPERIMENT, SPRING 1995

We conducted a "side by side" comparison of the impact of Pascal and Ada on the success of our CS 105 students. An instructor who was proficient in both Pascal and Ada was selected. Then we selected one "average" CS 105 section as the experimental group, which would use the Ada programming language. The primary control group was formed by the two Pascal sections taught by the experimental group's instructor. We also evaluated the performance of this control group against the Pascal sections taught by other instructors, to ensure that any performance differences could be attributed to the languages and not the instructor.

Every effort was made to minimize differences between the experimental and control groups. The students and lesson content were very similar in both groups. The section identified as the experimental group was "average." Students were not specially selected for this section. These students had "average" GPAs. A few students had some previous programming experience, but most did not. The same concepts and programming constructs were covered, lesson by lesson, with each group. For example, the second programming lesson focused on basic input and output for the keyboard and monitor. The control group learned about `read`, `readln`, `write` and `writeln`, while the experimental group learned about `get`, `get_line`, `put` and `put_line`.

4. RESULTS OF THE EXPERIMENT

The results of the experiment were instructive. We studied the impacts that a programming language made on achieving our course objectives. We determined that programming language choice affected students' understanding of software engineering constructs and their ability to grasp how real-world objects are represented in a computer. We also observed the impacts of syntax and development environments on novice programmer outcomes. We note that there were no significant differences between the performance of the Pascal sections taught by the experimental group instructor and the rest of the Pascal sections taught that semester. Therefore, we are confident that the results below can be attributed to differences in the two programming languages.

4.1 Early Support for Software Engineering

Ada supports the early introduction of software engineering concepts [6]. We found that use of library units communicated the concepts of encapsulation, abstraction and reuse very effectively to the experimental group. This was apparent as students wrote their first program using a simple output statement. Similar learning outcomes were not achieved in the control group.

In Pascal, input and output appear to students to be automatically provided by the compiler. The predefined Pascal subprograms `write` and `writeln` were easy to use. However, students in the control group did not readily recognize that Pascal input and output capabilities are actually predefined subprograms which were written by other programmers and must be linked with the student's code.

In Ada, standard input and output are encapsulated in the package Ada.Text_IO. The important learning outcome occurs because the students realize that they are using a predefined library unit and explicitly linking it into their program. Examination of the contents of the Ada.Text_IO

package reveals additional examples of encapsulation. Students see that the input-output routines for predefined types are encapsulated in generic subpackages within the Ada.Text_IO package.

Examination of this library unit is instructive in other ways. Students are able to conceptualize that a generic is a template for several analogous subprograms without having to dive into the implementation details too early. Students in the experimental group also learned that they could utilize Ada.Text_IO routines simply by studying the package specification without worrying about the implementation details in the package body -- abstraction at the most basic level.

Employment of the Ada library units also demonstrated an elementary example of reuse. It can be successfully argued that reuse at the library unit level is where most software reuse actually occurs.

Consider the two programs below:

```
WITH text_io;
PROCEDURE hello IS
BEGIN
      text_io.put_line(“hello world”);
END hello;
```

```
program hello;
begin
      writeln(“hello world”);
end.
```

The two programs are of approximately the same level of coding difficulty. Reading a package specification and elementary use of a library unit were not difficult for our students. Yet the Ada version provides more learning power as it forces the student to address abstraction, encapsulation and reuse at an appropriate introductory level.

4.2 How Computers Work

The first course goal of CS 105 is to become familiar with the fundamental concepts of computer science. While it is desirable to program at a high level of abstraction, it is also important for the students to understand what is “going on under the hood.” We observed that students in the Ada section did markedly better on their final examinations overall, but particularly in the area of computer science fundamentals. The Ada students seemed to have a much easier time understanding the differences among types, variables of those types and the real world entities they represent.

For novice programmers, implicit type conversions (as provided in Pascal) hide the fact that different data types are represented differently inside the computer. In contrast, the explicit type conversions required in Ada reinforce this concept at the cost of only a few keystrokes.

It is also important for beginning students to know something about the internals of the machines they are working on, and programming in Ada helps students appreciate this. Consider the difference between the mathematical concept of an infinite set of integers and the constraints imposed on the finite set of integers which can be represented by a machine architecture.

In Ada, a constraint error is raised when a programmer attempts to assign a value greater than 32,767 to a 16-bit integer. This makes it easy for novice programmers to recognize that an error has occurred, and to understand, at least at a conceptual level, what caused the error.

When this same error is committed in a Pascal program, the results can be very confusing to a novice programmer. For example, in Turbo Pascal for Windows Version 1.5, adding 5,000 to 30,000 results in a negative number (due to the arithmetic overflow resulting from exceeding the upper limit on a 16-bit integer stored using a 2's complement representation -- concepts which are not immediately obvious to a student who is just embarking on computer science studies). No error message is provided; the program simply produces the wrong result. If this is the final result, a student may notice this "strange" behavior. In that case, the student's initial reaction is usually disbelief that the computer is incapable of simple addition! If this addition is embedded in intermediate calculations, it might not even occur to a novice programmer to evaluate the addition as a source of the error. "How could it be possible that the computer cannot simply add two integers together? I don't even need to check that!" thinks the student who has not yet learned to differentiate between mathematical concepts and the *implementation* of these concepts on the computer.

4.3 Error Detection and Correction

Ada can make it easier for novice programmers to detect and correct errors. These features were evident at both compile time and run-time. Strong typing and concomitant type checking produce a compilation error if the problem can be detected at compile time. Otherwise, an exception is raised at run-time. The numeric overflow discussed above raises the exception, constraint_error.

Strong typing in Ada forces students to pay more attention to the significance of data types. As previously noted, this supports student understanding of data representation issues. This also makes it easier for students to detect problems early. We consider the arguments (excuses) for weak data typing to be largely without merit even for more

advanced programming students, and believe that such "features" clearly have no place in introductory instruction.

The power of the Ada exception mechanism manifests itself even in an introductory course. Our novices were able to reference their unhandled exceptions in the Ada Language Reference Manual and determine *why* their programs were failing.

4.4 Syntax Issues

Small -- but significant -- differences exist between the syntax of Pascal and Ada:

- Semicolon usage
- Inclusion of a block of statements in structured programming constructs
- Treatment of formal parameters

These differences had a critical effect on students' success.

Semicolons *terminate* statements in Pascal; they *separate* statements in Ada. Therefore, in Ada, semicolons are always placed at the end of every statement. In Pascal, a semicolon is placed at the end of a statement *unless* that statement is embedded in a special position in another statement which you do not want to terminate immediately. For example, the Pascal statement "x := y * 5.0; " would normally include a semicolon. However, if that same statement is located as shown below, then it must *not* be followed by a semicolon:

```
if  (x < y) then
   x := y * 5.0
{note: semicolon forbidden; still inside if statement}

  else
    x := y * 2.0;
{note: semicolon required; terminates if &
                              assignment
statements}
```

This seemed inconsistent to the students in the control group, who often inadvertently terminated an `if` statement before its `else` clause with one misplaced semicolon. This problem was exacerbated by the insufficiently specific error message ("Error 113: Error in Statement") provided by the Turbo Pascal compiler under these circumstances.

Students in the experimental group also benefited from Ada's clear syntax for identification of a block of statements to be included inside a structured programming construct (e.g., IF, CASE, WHILE, FOR). In Ada, reserved words delimit the different parts of a structured programming construct. For example, here is the Ada syntax for a while loop:

```
-- reserved words shown in upper case

WHILE boolean_condition LOOP

       statement; -- repeat as
                  -- neccesary

END LOOP;
```

Students may include as many statements as needed between "loop" and "end loop." The compiler provides a useful (i.e., suitably specific) error message if a student omits one of the required reserved words which serve as delimiters for the while loop.

In Pascal, the syntax for a while loop is as follows:

```
while  ( boolean_condition ) do
   statement;
```

If a student needs to include more than one statement in the while loop, he or she must construct a compound statement by "bracketing" the statements to be accomplished inside the loop with a `begin` and `end`. If the student omits the `begin` and `end`, the code will still compile – but only the first statement will actually be executed inside the loop. All remaining statements will be executed exactly once, after the loop has terminated. This tends to produce output which is very different from what the programmer intended, and it can be a very difficult error for novice programmers to detect.

The use of compound statements in Pascal, compared with delimitation using reserved words in Ada, is also an issue with selection constructs (if, case). Omission of the begin/end pair for selection constructs causes compile-time, rather than run-time, errors. While these errors are easier to detect and correct than the run-time errors which occur with repetition constructs (while, for), they still prevented many of our control group students from completing quizzes and graded labs. This was due, in part, to the cryptic "Error 113: Error in Statement" message provided by the Turbo Pascal compiler, and we admit that a better error message might have helped the control group find this error more quickly for selection constructs. We note, however, that even a better compile-time error message would not resolve the difficulties generated when the begin/end pair is omitted in repetition constructs.

We observed significant differences between the abilities of the experimental and control groups to pass data between procedures using parameters, and we attribute these differences to language syntax.

In Ada, the rules governing formal parameters, including the specification of their parameter passing mode and their use inside procedures, are very clear and are enforced by the compiler. Consider the following Ada procedure specification:

```
PROCEDURE DoSomething
       ( par1:IN integer;
         par2:IN OUT character;
         par3: integer);
```

Par1 is explicitly identified as an "in" mode parameter, meaning that it is intended to be read but not changed. Par3 is also an "in" mode parameter, because its mode is not explicitly specified and "in" is the default. Any attempt to write to an "in" mode parameter will cause a compile-time error. Therefore, if a student simply forgets to specify par3's parameter passing mode and subsequently tries to change that parameter's value inside the procedure, the compiler will notify the student of this error. Students in the experimental group readily understood this concept, and were able to quickly correct any errors in their parameter passing modes.

In Pascal, parameter modes are much more confusing to novice programmers. A parameter's value can always be changed inside a procedure. However, that value is not passed back out of the procedure unless the parameter was declared as a "`var`" parameter – and the syntax required to do this is easily misunderstood by novice programmers. Failure to include the "`var`" (either because it was omitted completely, or because a student did not understand that the "var" must be repeated after each semicolon which appears in the parameter list) can lead to erroneous program behavior that is very hard to debug. Consider the following Pascal procedure header:

```
procedure DoSomething
    (par1: integer;
     var par2:char;
         par3:integer);
```

The values of all three parameters may be changed inside procedure DoSomething. However, only the new value for par2 will be returned to the calling routine. The new values of par1 and par3 will be discarded, without any notification to the programmer. We found this difference especially critical for a CS 0 course such as CS 105, in which students lack the depth of understanding needed to track down such subtle errors. This particular problem caused many difficulties in our Pascal cadets' second and third graded labs.

4.5 Environment Comparison

A major concern at the outset of the experiment was that, regardless of the merits of Ada, the available environments might be too hard for novice programmers to use. If novice students are unable to successfully work in the environment, the merits of the language are immaterial. Our students had no difficulty working in the then available Ada environment. PC-based Ada environments now are even better. One environment of particular interest is AdaGIDE which was developed at the United States Air Force Academy [3].

The control group used Turbo Pascal for Windows 1.5. The experimental group used Meridian OpenAda 2.0 (an Ada 83 environment.) The Turbo Pascal environment was easier to use in many ways than the OpenAda environment. However, the ease-of-use features of the Turbo Pascal environment did not facilitate the desired learning outcomes. The user-friendly Pascal environment hid the details of the compile -> link -> run steps. This blurred the distinction between source code, object code and executable code. This obfuscates the true nature of a computer. Many students in the control group were unable to understand how the code they wrote was executed by the machine.

The experimental group adapted quickly to the Meridian OpenAda environment. Students were required to connect their working directory to the Ada library units. Then they were able to compile, then link and then execute their code. Each step was simple but distinct. This process explicitly demonstrated to the students how their source code was transformed into an executable program.

5. CONCLUSIONS

We were pleased to see the very positive effect of the Ada programming language on the success of novice programmers. As a direct result of this study, which demonstrated that our students could go farther, faster in Ada than they could in Pascal, the Academic Board of the U.S. Military Academy approved changing to Ada 95 as the programming language for all students in our introductory computer science class. We note that these results have been independently achieved and verified at the U.S. Air Force Academy [4]. Other programming language analyses conducted in upper division courses have produced similar results, showing the pedagogical advantages of Ada 95 [2].

This experiment gave us an excellent opportunity to isolate the effect of a programming language on novice programmers. Programming language decisions are often complex because a programming language is just one factor associated with any software effort [5]. We submit that the programming language study conducted at West Point is a model approach for determining the comparative pedagogical advantages of programming languages and offers a sound basis for an unbiased decision.

6. ACKNOWLEDGEMENTS

Thanks to Major Thomas Crabtree, U.S. Army, who played a critical role in the conduct and support of this experiment.

7. REFERENCES

[1] Ben-Ari, M. and Henney, K., "A Critique of the Advanced Placement C++ Subset," *Special Interest Group on Computer Science Education Bulletin*, Vol. 29, No. 2, September 1991, pp. 7-10.

[2] Blair, J.R.S., Ressler, E.K., Wagner, T.D., "The Undergraduate Capstone Software Design Experience," *Tri-Ada '97 Proceedings*, Nov 9 -13, 1997, St. Louis, Mo., pp .41 - 47.

[3] Carlisle, M. C., and Chamillard, A.T., AdaGide: *A Friendly Introductory Programming Environment for a Freshman Computer Science Course*, 11th Ada Software Engineering Education Team Symposium, Monmouth Univ., Monmouth, N.J., 12-13 Jun 97.

[4] Chamillard, A.T., and Hobart, Jr., W.C., "Transitioning to Ada in an Introductory Course for Non-Majors," *Tri-Ada '97 Proceedings*, Nov 9 -13, 1997, St. Louis, Mo., pp. 41 - 47.

[5] Computer Science and Telecommunications Board, National Research Council, *Ada and Beyond, Software Policies for the Department of Defense*, National Academy Press, Washington, D.C., 1997.

[6] Feldman, M. and Koffman, E., *Ada 95: Problem Solving and Program Design*, Addison-Wesley, Reading, Mass. 1996.

[7] Hamilton, J.A., Jr., "Why Programming Languages Matter," *Crosstalk*, vol. 10, no. 12, December 1997, pp. 4 - 6.

[8] Hamilton, J.A., Jr., Cook, D.A., "Ada Training and Education in the US Army and US Air Force," *Tri-Ada 96 Proceedings*,. Dec 3 - 7 1996, Philadelphia, Pa., pp. 151 - 155.

[9] Mody, R.P., "C in Education and Software Engineering," *Special Interest Group on Computer Science Education Bulletin*, Vol. 23, No. 3, September 1991, pp. 45-56.

[10] Reid, Richard J., "First-Course Language for Computer Science Majors," Internet Survey, ftp.cps.msu.edu:pub/arch/ CS1_Language_List.Z.

[11] Suchan, W.K., Smith, T.L., "Using Ada 95 as Tool to Teach Problem Solving to Non-CS Majors" *Tri-Ada '97 Proceedings*, Nov 9 -13, 1997, St. Louis, Mo., pp. 31 - 36

Lieutenant Colonel J. A. (Drew) Hamilton, Jr., Ph.D., (dj7560@eecs1.eecs.usma.edu) is Research Director and Assistant Professor in the Department of Electrical Engineering and Computer Science at the United States Military Academy, West Point, NY. Previously, Colonel Hamilton was Chief of the Ada Joint Program Office. http://www.eecs.usma.edu/usma/academic/eecs/instruct/hamilton

Major Jeanne L. Murtagh, (jmurtagh@afit.af.mil) is Director, Software Professional Development Program (SPDP), at the Air Force Institute of Technology, Wright-Patterson, AFB, OH. Major Murtagh previously served as an Assistant Professor at West Point where she directed and implemented the results of the USMA language study.