WHY ADA® IS NOT JUST ANOTHER PROGRAMMING LANGUAGE

Ada's importance goes far beyond its initial limited goals. The worldwide interest in the usage of Ada is one of many reasons for its uniqueness.

JEAN E. SAMMET

INTRODUCTION

General

Ever since the advent of Ada®—regardless of the date one might choose for that event—its detractors have tried to dismiss it as *"just another programming language."* This statement is generally intended to indicate that there is no good reason for so much activity, excitement, or work in Ada. Although it is certainly true that Ada is indeed one of hundreds of programming languages, this author's contention is that Ada has distinctive technical and nontechnical characteristics that justifiably continue to spur on the large "cult" that has sprung up around it. It is the combination of (1) the specific language features, (2) some auxiliary technical aspects, (3) the process that developed and supports it, and (4) its acceptance by an international computing community of managers, users, researchers, and governments that together make Ada unique.

The purpose of this article is to indicate Ada's uniqueness in the panoply of current and past high-level programming languages. In addition to the introductory material, the article is divided into two major sections. The first deals with the nontechnical aspects creating Ada's uniqueness, and the second involves a more technical discussion. Although this may seem like a somewhat unorthodox approach, it is in fact the nontechnical characteristics of Ada that are even more important than the technical features of the language in determining why it is *not just* another programming language.

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

Introductory Comments on Ada

An unparalleled *combination* of three specific technical elements sets Ada apart from other languages: Ada is indeed a programming language, but is also much more since it offers major support for software engineering principles, and it includes many aspects of a programming environment.

Ada's method of development and the strong interest in it—even internationally—beyond just the community responsible for its development (i.e., the Department of Defense) are crucial factors in terms of its uniqueness. Ada is unique due to sociological, economic, and political reasons—nontechnical issues not traditionally applicable or considered heavily in the evaluation of other languages. Nevertheless, they are crucial to an understanding of the role that Ada plays in the software world of today.

Ada's technical uniqueness is based on its support for the concept of software components, its excellent blend of modern useful features, and its support for the production of very large software systems. In addition, there are various administrative controls on the language and its implementation that collectively are specific to Ada alone. These and other issues pertaining to the implementation, aspects of the environment, and other miscellaneous factors that support my basic contention are discussed in more detail throughout the course of this article.

NONTECHNICAL REASONS WHY ADA IS NOT JUST ANOTHER PROGRAMMING LANGUAGE

History and Method of Language Development

Early Development. To understand the unique nature of the development of Ada, it is necessary to be familiar with its early history. (More details can be found in [7, 16, 24].)

In 1974 a report estimating the future costs of Department of Defense (DoD) software development was produced; the projected cost of over $3 billion annually was considered horrendous. In addition, studies also showed that there were hundreds of languages and/or their dialects being used by the Defense Department and its contractors, which obviously made it difficult to interchange programs or programmers and made it harder for DoD to maintain software.

In 1975 Lieutenant Colonel William Whitaker, then assigned as Military Assistant for Research in the Office of the Director of Defense Research and Engineering (DDR&E), reviewed plans from the military services for additional funding to further develop their separate existing languages (e.g., JOVIAL and CMS-2). He felt that it was worth investigating whether a single high-level programming language could meet the needs of all DoD military application software. Although an affirmative answer seems quite obvious *now*, it was *not* obvious *then* from either a technical or political viewpoint. A High Order Language Working Group (HOLWG) composed of representatives from the military services and defense agencies was established to oversee this investigation.

In 1975, Dr. David Fisher at the Institute for Defense Analyses (IDA) was asked to create a draft requirement for a high-level language that would meet the programming needs of the Department of Defense embedded computer systems. (An embedded computer is a computer that is part of a larger—probably noncomputer—system, which need *not* necessarily be a weapon. Examples include missiles, submarines, and airplanes, but also automobiles and microwave ovens, as well as general applications such as air traffic control.) Dr. Fisher produced a document of requirements called "STRAWMAN," which was sent to many interested people in both the military and civilian communities. Comments were received on this document, and a revised version called "WOODENMAN" was issued late in 1975, followed by a third set of requirements labeled "TINMAN" issued in 1976. During 1976, 23 languages then in existence were evaluated against the TINMAN requirements. (The list of languages evaluated is shown in Table I.) It was not entirely surprising that no existing language met all the requirements expressly stated in the TINMAN document and implicitly stated by the DoD community. This evaluation did conclude that it *would* indeed be possible to construct a single language to meet the requirements. Three languages—ALGOL 68, Pascal, and PL/I—were deemed adequate as baselines for starting to create a language that would meet the TINMAN requirements. (It was *not* intended that the new language be upward compatible from the baseline.)

The importance and uniqueness of this process of producing requirements, evaluating them based on public commentary, revising them, and then repeating the cycle is often underestimated in terms of "requirements" and "public commentary" (discussed in more detail later). The existence of a clear statement of requirements prior to developing language specifications is unique. All previous languages have had short or general or vague or nonexistent requirements or objectives. Even when requirements did exist, they were generally intertwined with the actual language design. In Ada's development, however, the language design occurred only after numerous refinements to the requirements had been made.

After the evaluations against TINMAN were finished, the Department of Defense undertook in 1977 another activity unique in language development. They issued contracts to four vendors (out of about

TABLE I. List of Languages Evaluated Against TINMAN Requirements in 1976

Used for DoD and/or Embedded Computers
CMS-2
CS-4
HAL/S
JOVIAL 3B
JOVIAL J73
SPL/1
TACPOL
Used for Process Control and/or Embedded Computers (primarily in Europe)
CORAL66
LIS
LTR
PEARL
PDL2
RTL/2
Research-Oriented Languages
ECL
EL-1
EUCLID
MORAL
Widely Used and/or General Languages
ALGOL60
ALGOL68
COBOL
FORTRAN
Pascal
PL/I
SIMULA67

15 bidders) to create preliminary language designs based on an updated set of requirements called "IRONMAN," again developed by Fisher. At the end of a six-month period (i.e., in January 1978) each of the submitted design documents was stripped of its developer's identification and given a color code: Blue, Green, Red, or Yellow. The four sets of "anonymous" language designs were sent to anyone who cared to review them, and specific groups were commissioned by each military service to examine the results from that service's viewpoint. (Although there was an honest attempt to hide the source of each particular language design, the information did not always remain secret.) In 1978, Intermetrics (in Boston) and Cii-Honeywell Bull (in France), whose language designs had been designated Red and Green, respectively, were chosen to do further language design based on Fisher's final set of requirements, now called "STEELMAN." (A listing of the topics covered in STEELMAN, and a few specific examples, are shown in Figures 1a and 1b [9].) In 1979, the final language designs from those two vendors were sent out for public commentary and the Green version was accepted, labeled "Preliminary Ada," and published in *ACM SIGPLAN Notices* [10]. Also published with the language specifications was a unique document called "Rationale," which described the reasons for many specific language features. It is primarily because the DoD requirements are themselves large that the language is large. The DoD needs run the gamut from fixed point arithmetic to concurrent processing to support of modern software engineering concepts.

The significance of this development process is that no programming language ever created underwent this amount of design competition, and in fact I cannot recall any language that had *any* competitive procurement. It is important to note that although the funding for all this language development came from the U.S. Department of Defense, the language was actually created by a group physically located in France.

Test and Evaluation. Most languages, after their development, are thrust upon their intended user community, which then either accepts them, rejects them, or makes random suggestions for improvements. The implementers also provide large inputs. For example, the first three sets of official COBOL specifications (i.e., COBOL 60, 61, and 61 Extended) were modified in quick succession based on user and implementer feedback. PL/I's early specs were publicly issued a few months apart (i.e., March, April, and June 1964). Since Pascal was created by one person, there was very little change from the first version to the second, and then nothing official until the ISO and ANSI standards. This "normal" but very haphazard process contrasts with the actual methodology used in Ada. From the very beginning of the activity, a *formal* test and evaluation phase was planned. Approximately 80 teams of programmers used "Preliminary Ada" to write portions of systems they had previously coded in other languages and suggested language changes based on their experience. Most of these participants were people who intended to eventually use the language. A workshop with presentations and discussions of many of these evaluations was held in October 1979. The participants and other language experts also submitted over 900 detailed comments that were initially analyzed by a company under contract to DoD and then by a group of language experts called the

1. General Design Criteria
2. General Syntax
3. Types
   - 3.1 Numeric Types
   - 3.2 Enumeration Types
   - 3.3 Composite Types
   - 3.4 Sets
   - 3.5 Encapsulated Definitions
4. Expressions
5. Constants, Variables and Scopes
6. Classical Control Structures
7. Functions and Procedures
8. Input-Output, Formating [*sic*] and Configuration Control
9. Parallel Processing
10. Exception Handling
11. Representation and Other Translation Time Facilities
12. Translation and Library Facilities
13. Support for the Language

FIGURE 1a. STEELMAN Requirements Topics

3C. Type Definitions
It shall be possible to define new data types in programs. A type may be defined as an enumeration, an array or record type, an indirect type, an existing type, or a subtype of an existing type. It shall be possible to process type definitions entirely during translation. An identifier may be associated with each type. No restriction shall be imposed on user defined types unless it is imposed on all types.
3-1F. Integer and Fixed Point Numbers
Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.
6B. Sequential Control
There shall be a control mechanism for sequencing statements. The language shall not impose arbitrary restrictions on programming style, such as the choice between statement terminators and statement separators, unless the restriction makes programming errors less likely.

FIGURE 1b. Examples of Requirements from STEELMAN

"Distinguished Reviewers," who provided judgments on these comments to Dr. Jean Ichbiah, the leader of the language team. Because of the timing there were no practical implementations of "Preliminary Ada" and so the test and evaluation work could only be deemed a paper exercise. Nevertheless it produced some valuable suggested changes in the language in a controlled environment.

Public Commentary. Perhaps the most striking characteristic of the Ada development was DoD's positive emphasis on, and encouragement of, massive public commentary from academic and industry experts here and abroad as well as from DoD employees and consultants. This public commentary was solicited throughout the development of the requirements documents and both phases of the competitive language design, on the "Preliminary Ada" specifications via the formal Test and Evaluation phase, and, of course, on the proposed ANSI standard. There is nothing unique about comments in the ANSI phase, but in no other language development was this massive public commentary solicited *prior* to the standardization phase. In the case of PL/I, the development committee made several reports to SHARE (an IBM user's group) about the early language specifications, but received only haphazard response.

This commentary process was enhanced by encouraging the use of the ARPANET for electronic mail. It seems unlikely that all the material could have been handled without the use of computers for correspondence analyses and tracking. This is unique since I believe only the research language EUCLID made extensive use of electronic correspondence.

Language Design Decision Making. Most of the languages developed in the past have been created either by a committee or by a single person. COBOL is a prime example of the former whereas Pascal and APL illustrate the latter. In the case of Ada, there was a language design *team* (not a committee) based primarily in France working for the Cii-Honeywell Bull company but augmented by team members from other countries and other companies. Although this truly was a team effort, a single person, namely Dr. Jean Ichbiah, either made the final design decisions himself or delegated that authority in some specific cases. In this context it is important to understand that although the work was done under a contract with the Department of Defense, the DoD did *not* mandate specific language decisions.

Strong Interest Outside the Department of Defense Community

Although designed to meet the requirements of a particular user community—namely Department of Defense embedded computer systems—Ada is unique because of the strong interest it engendered in many places *outside* its developing organization. Although there have been other languages created under DoD auspices—e.g., JOVIAL, CMS-2, and TACPOL—none of these languages or their multitudinous versions caused the excitement and interest that was evidenced in Ada almost from its inception. Coverage of these languages in the technical journals has been sparse; for Ada, however, there were literally thousands of comments from all over the world during the phases described earlier. People from numerous universities—both large and small, major and minor—were involved. In addition to the numerous DoD contractors, other industrial companies as well as many software houses also participated.

There has been (and still is) strong interest in Ada from both the European military and commercial sectors. In fact, computer scientists from numerous organizations in Europe directly participated in various aspects of the technical and administrative development. The EEC (European Economic Community) and NATO have both adopted Ada for use and research.

Not since ALGOL 58 (and to a leser extent ALGOL 60) has there been so much initial, widespread interest in a language outside its developing organization. Ada is also the only programming language that has ever been a line item in the Federal budget and garnered significant Congressional interest.

ACM SIGAda. An indication of the widespread interest in Ada is the Technical Committee that was formed under ACM SIGPLAN in 1981 and known then as ACM AdaTec. It actually grew from an informal group of implementers that started meeting in 1980 and expanded greatly thereafter. In 1984, AdaTEC became a full-fledged Special Interest Group (SIG) under ACM and by June 1986 it had over 3800 members. Since this group is completely independent of Department of Defense control, it exemplifies the widespread interest in this particular topic. SIGAda and SIGAPL are the only two specific language SIGs in ACM. It is also worth noting that formal Ada language conferences sponsored by ACM in 1980 and 1982 attracted over 500 people each, and in 1985 the ACM SIGAda International Conference held in Paris was attended by 415 people from the U.S., Europe, and Asia. Furthermore, for several years there have been quarterly meetings either organized under SIGAda auspices, or involving their cooperation, and attended by hundreds of people (or as many as a thousand). No other language—and in fact no technical topic that I can think of—has had quarterly public meetings attracting many hundreds of attendees (including many from outside the U.S.) over a period of several years.

Applications and Use Outside DoD. If Ada's use were limited to DoD applications it would not have achieved as much importance and gained so much attention, and it would indeed be "just another programming language." But there has been considerable interest in Ada's practical use from outside the DoD. As far as I know, the first significant application of Ada—programs written for profit and not for testing or evaluation—was prepared by a small software company called Intellimac in the metropolitan Washington D.C. area for a commercial trucking company [8]. The applications were standard classical business data processing activities such as payroll. Thus, at a very early stage Ada demonstrated its applicability for commercial use on *non*embedded computer systems. As discussed later, there is nothing in the technical characteristics of Ada that makes it applicable only to embedded systems—even though that application motivated the original requirements and language design.

Of course there are numerous types of embedded computers in major use outside the Department of Defense—e.g., automobiles, microwave ovens, robots, and process control, as well as applications coming from the FAA and NASA. It is not the purpose of this article to explain all of those uses but merely to indicate that they have existed and will continue to do so. It is, however, interesting to note that one of the papers at the first Ada conference not organized by the DoD, namely the ACM SIGPLAN meeting in December 1980, dealt with the use of Ada for programming involving a microwave oven.

Additionally, Ada has been used as a significant vehicle for research and development; much of the material published in the professional language journals uses Ada as a base or involves Ada in some way, which has not occurred *to this extent* since ALGOL 58 and ALGOL 60.

Economic Issues. Ada is the first language to inspire the creation of many companies whose sole business is Ada technology and support. Some of them concentrate on compilers and/or other tools, while others have gone into business to provide Ada education or even computers whose design is based on Ada. Although there is no widespread practical occurrence yet, there has been considerable talk about creating saleable software components. A software repository of tools is now available on the ARPANET. It is important to note that the endeavors in the private sector have been strongly encouraged by the Department of Defense.

Language Control

Most languages created in the past have had either no control over their early growth and changes, (e.g., FORTRAN, JOVIAL) or very strong control (e.g., COBOL). The Department of Defense has tried to work in a middle ground between these two extremes, and their main tools for doing so have been the policies of *trademark* and *forbidding subsets or supersets.* The early ANSI standardization also prevented dialects. The DoD policies regarding validation (discussed in a later section) have also helped control the language.

Trademark. The only two languages trademarked in the past that I am aware of are TRAC® and SIMSCRIPT®. Ada has been trademarked since 1981; the main purpose of the trademark is to prevent compilers that do not conform to the language standard from being sold as true Ada compilers.

Forbidding Subsets or Supersets. One of the most hotly debated issues in earlier days of Ada had to do with subsets. The Department of Defense took a very strong position that they simply would not allow subsets and no subset compiler could use the name Ada. They allowed exceptions in the early stages, when implementations were just getting started and it was useful to have even subset compilers publicly available. A number of people, myself included, objected to that policy as being impractical. It seems fair to say, however, that experience has proven both the DoD *and* the "objectors" correct. In the early stages every compiler implemented a different subset because vendors were anxious to produce something for users to try, and they had neither the time, the knowledge, nor the resources to implement all of Ada initially. On the other hand, the DoD has "stuck to its guns" and refuses to allow any compilers implementing only a subset to be recognized as full Ada compilers, that is, the subset must be identified in the advertising.

By refusing to allow unilateral language extensions to be implemented in any Ada compiler, the DoD significantly increased the likelihood of achieving the software portability that is one of Ada's major objectives. A compiler that implements extensions to Ada will not be validated. Almost all other standard languages allow compilers that provide additional facilities and hence prevent portability. The only language developer I can recall that took such a rigid position was Professor Victor Yngve (MIT) who developed COMIT (a string processing language) in

® TRAC is the trademark and service mark of Rockford Research Institute, Inc.

® C.A.C.I., Inc.

1957. It was eventually superseded by SNOBOL—partly because SNOBOL seemed better and partly because users resented their inability to "fiddle" with the language.

Early Standardization. By 1986, eight other programming languages had been standardized by ANSI and/or ISO: ALGOL, APT, BASIC, COBOL, FORTRAN, MUMPS, PASCAL, and PL/I. In each case the work on standardization did not start until there were full implementations and some significant experience. Ada, however, was quite different. Almost as soon as "Preliminary Ada" had been selected, the Department of Defense started to investigate the processes by which Ada could become an ANSI standard. The DoD recognized that in doing so they might lose some control of the language to the wider community, but felt that it would be more beneficial to the DoD in the long run to have an ANSI standard as early as possible. After much discussion between the DoD people and ANSI and its Sectional Committee on Information Processing X3 (which has handled the United States standardization of almost all other programming languages) it was decided that Ada would be standardized under ANSI auspices by using the canvass method. The only other language standardized by that method was MUMPS, a language developed within the hospital community; all other languages involved open committees.

Standardization is not an easy process for any language, but the number of public comments for Ada certainly exceeds all those submitted for other languages. According to Dr. Jean Ichbiah, [18, p. 991], "... my group had to review 7000 comments produced by experts from 15 different countries.... " After the standard was adopted in February 1983 [3], a group had to be established to deal with interpretations and, potentially, a revised standard. The DoD, as part of its agreement with ANSI, has established an Ada Board to perform these functions, which are normally handled by X3 Technical Committees. To complete the picture, although it is certainly *not* unique, the International Standards Organization (ISO) has accepted the ANSI Ada standard as a draft standard and has started the final balloting to determine whether it should become an international standard as well. Final results will not be known until late in July 1986, but no problems are anticipated.

Implementation Facts and Issues

Compiler Validation. Although compiler validation is not new, it has been handled in a unique way for Ada. In the other cases of compiler validations, the validation process was instituted long after the language had been created. For example, COBOL validation work started as early as 1963 but was not made mandatory by the government until 1975, and it was the first language with such a facility. Validation facilities for other major languages (e.g., FORTRAN, BASIC, and PASCAL), were developed even later.

In the case of Ada, the compiler validation facility was planned from the very beginning as an integral part of the effort [17]. Drafts of tests were made available to the early implementers for comment; in fact, the creation of these test drafts helped find ambiguities in the language. The tests are publicly available to anyone who wishes to obtain them. As part of the validation process, the implementers must successfully run the tests themselves before applying for official validation tests to be run on their premises. The tests are periodically upgraded; new test suites were released on a six-month cycle until 1986, when a change to a yearly cycle was made. DoD has stated repeatedly that it will allow only validated compilers on its projects. This of course does not require any non-DoD user to require a validated compiler, but most of them will do so to achieve the benefits of knowing the compiler conforms to the standard. As of June 1986 there were 47 validated compilers, on over 30 host-target-operating system triplets from 19 vendors/developers.

It should be emphasized that—as with all validation tests—the intent is only to measure conformity with the standard. Any validated compiler may still have bugs and poor performance, since performance itself is not measured by the validation tests. On the other hand, significant work is being done by SIGAda to provide programs that measure performance.

Environment. One of the unique features of Ada was the recognition from the very beginning of the need for a proper software development environment to achieve the full benefits of the language. Anyone involved in developing software knows that a language and its compiler are not sufficient to provide adequate facilities for the effective use of the language—regardless of which language it is; there is a need for tools and supporting software as well. After some early false starts, the Department of Defense indicated the need for an Ada Programming Support Environment (APSE). As with the language, a series of requirements documents were issued with allowance for public commentary. An early version was named "Pebbleman" and after a few intermediate steps, the final set of requirements documents for APSE was labeled "STONEMAN" [11, 14].

The basic concept proposed in STONEMAN is the existence of three levels: KAPSE, MAPSE, and full APSE. The KAPSE (Kernel APSE) is the level just above the native operating system and is perceived to be the interface between the operating system and anything else. The MAPSE is intended to be a Minimal Ada Programming Support Environment and would normally contain facilities such as the compiler, a linker or loader, a debugger, an editor, etc. The full APSE would include other tools needed to support applications written in Ada. The STONEMAN concept is represented by the diagram in Figure 2 [11].

The very early DoD implementation plan assumed that one contractor would produce a rehostable/retargetable compiler and an associated support environment. In actuality, DoD issued contracts to two vendors, who each produced a compiler and a MAPSE on differing major computers. Although a major goal of the DoD throughout the Ada effort was to get portable application programs and tools, it was immediately clear that the two different environments might limit portability. To study and solve this problem, in 1982 the DoD created the KIT (Kapse Interface Team) and the KITIA (KIT From Industry and Academia) to develop a common interface from the level just above the native operating system. After several years of investigation this led to the creation of the Common APSE Interface Set (CAIS), which is a set of Ada packages that define interfaces to support the development of transportable tools written in Ada [12]. The CAIS is a separate story; its importance in the uniqueness of Ada is that the environment has been, and continues to be, considered in parallel with the language.

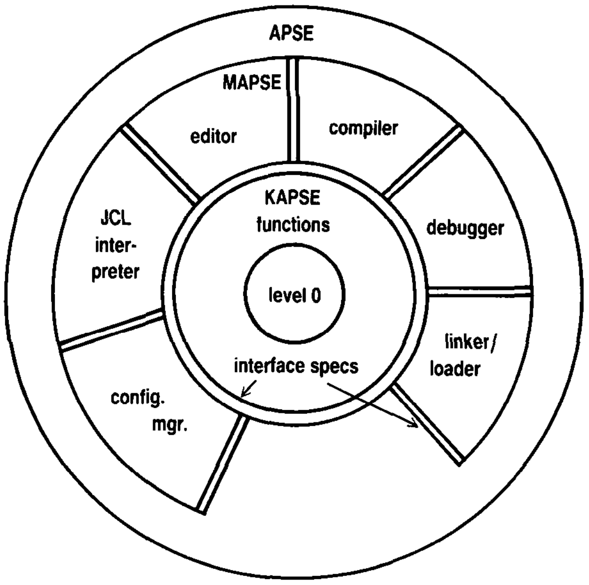

FIGURE 2. Representation of STONEMAN Level Concept

Miscellaneous Nontechnical Reasons for Ada Not Being "Just Another Programming Language"

The miscellaneous nontechnical reasons why Ada is not just another programming language shown here are not necessarily in order of significance or chronology.

STARS. Ada is considered the cornerstone of Software Technology for Adaptable, Reliable Systems (STARS). This was initially known as the DoD Software Initiative, and is the attempt by the Department of Defense to make a significant improvement in software development and its methodology. An early description of the intent is in [15] and the current status is in [22].

Use as a Design Language. Ada is being used as a design language in many different ways and is required by many DoD projects. Prior to Ada's existence, there appeared to be no significant attempts at using a *specific* high-level language as a design language, although various pseudocodes were developed. Although my colleagues and I were among the pioneers in using Ada as a design language, we were by no means the only group doing so. An examination of almost any issue of *Ada Letters* will show some work in this area, and several times a year it prints a list of organizations doing similar work. An attempt was started in 1982 by the IEEE Computer Society to standardize this new technology; their effort is still underway despite protests from many knowledgeable people that it is premature to try to create a standard, or even a recommended practice.

Computer Architecture. Ada has influenced computer architecture—at least technically. Carnegie-Mellon University, under contract to the Army, designed a computer architecture around 1980–81 known as NEBULA that was intended—among other objectives—to be effective for the translation and implementation of Ada programs. The INTEL iAPX 432 was originally claimed to be influenced in its hardware design by Ada. A more current illustration is the computer (and related tools) built by the Rational Corporation to support development of Ada programs.

Languages other than Ada *have* influenced hardware architecture, namely ALGOL, FORTRAN, SYMBOL, and LISP. But only LISP comes close to having a practical effect.

Use in Education. One of the key aspects of Ada is its usefulness for new types of software education. Ada has been, and should be, used as a vehicle for teaching software engineering principles in both academic and industrial settings. (This is further discussed later in the article). It can be used to teach good software design, and has also been used in a number of universities to teach specialized topics such as numerics, concurrent processing, and data structures. In some places it is taught as a first language.

Pascal certainly preceded Ada in having a significant effect on education and is widely taught as a first language in many colleges and universities. Pascal does not, however, allow for coverage of many of the specialized topics that Ada does.

All *good* courses in Ada cover far more than the language syntax, and usually emphasize its role in software engineering (discussed in the next section).

ADA IS UNIQUE TECHNICALLY

In trying to determine the actual uniqueness of Ada from a technical viewpoint there are a number of issues to examine. Ada has a large set of specific important features that appear in an integrated way, although the features themselves are not necessarily unique. Most of them have appeared previously in some other languages but it is their integration that is important.

The major technical characteristic of Ada is that it supports most modern software engineering principles, and in particular it is designed around the concept of the software component. Furthermore, Ada supports the production of very large software systems, which was a major design goal. These points will be discussed more fully in later sections, along with indications of where other languages fit into the overall pattern.

In the context of this article it is only possible to provide a smattering of technical information on Ada. The list of references at the end contains sources of many kinds, ranging from the full 300-page Language Reference Manual [3] to a Self-Assessment Procedure [23]. At least 50 books on Ada exist. *Ada Letters* frequently prints lists of references [1].

Some Technical Highlights

Each person writing about Ada has their own favorite list of Ada's important technical features, and this author is no exception. Probably the only feature that almost everybody agrees is the most important is the "package," which will be discussed later. The key technical features are the following, listed in no particular order.

- packages
- strong data typing
- generics
- tasking
- numeric processing
- real-time processing
- exceptions
- overloading
- separate compilation
- representation clauses

Software Engineering and Ada

One of the key characteristics that makes Ada not just another programming language is its adherence to, and support of, most software engineering principles. As with the list of technical highlights, each person has their own view of the important software engineering principles. The list below is provided for use as a baseline for demonstrating Ada's support of software engineering. There is no particular significance to the sequencing and certainly the reader should not debate the merits of this particular list; Ada and software engineering are the important concerns in this context.

- structured programming
- top-down development
- strong data typing
- abstraction (of data and actions)
- information hiding and encapsulation
- separation of specification from implementation
- reusability
- separation of logical from physical concerns
- portability
- modularity
- readability
- verifiability

Consideration of Ada and Specific Software Engineering Principles

Structured Programming. There is certainly nothing unique about the fact that Ada is useful for structured programming, as many languages have the associated facilities. For the record, I point out that Ada has a normal *if* . . . *then* . . . *else* . . . structure and *begin-end* bracketing for blocks. Loops are handled by *for* . . . , *while* . . . , *exit when* . . . constructs.

Top-Down Development. There is also nothing new about Ada's ability to handle this, but the facilities are somewhat different. Various subunit concepts (specifically packages), as well as the data abstractions, make it easy to do top-down development and allow the creation of stubs. In addition, separate compilation can be used to combine portions of a program which were developed separately in a top-down fashion. This makes Ada very effective for producing large software systems.

Strong Data Typing. Ada requires that each data object have a type declaration; there are no implicit defaults. In addition, Ada is the essence of strong data typing because it forbids different types of data to be mixed; there must be explicit type conversions executed before combining them. Scoping rules also enforce strong typing.

This is actually a much older concept than people may realize. The initial FORTRAN of 1957 had the essence of strong data typing in forbidding the programmer to write "$A + I$" because default conditions automatically declared A as a floating point number and I as an integer; FORTRAN permitted implicit conversion (e.g., $I = A$). The essence of weak data typing appeared shortly thereafter in COBOL, which allowed the combination of almost any two data items, and the compiler was expected to convert correctly. This weak typing was carried to even greater extremes in PL/I where many pages in the manual were devoted to the rules for combining *un*like types. In looking at newer major languages, PASCAL again reversed the trend by reinstituting an interest in strong data typing that was of course carried to even greater extremes in Ada. In fact, Ada even supports the strong typing across separately compiled units.

Abstraction (Data and Action). Abstraction of actions—that is, an indication of what is to be done without necessarily indicating the details—is achieved in four ways in Ada, namely by functions, procedures, packages, and generics. The first two of these are naturally quite old and have no particular novelty as far as Ada is concerned, but the package itself is another matter. Packages can contain just data types and descriptions (as in the early JOVIAL COMPOOL and the COBOL Data Division) or just functions and/or procedures, or in the most normal situation, a combination of both. As was indicated earlier, this is probably the strongest new technical feature of Ada.

In previous languages there was no way to separate the specification of what was to be implemented from the body of the implementation. In something as simple as a square root, there is no formal way in any language to define anything about the variable whose square root is to be taken; the very important fact that the variable must be positive is buried in the code itself. Ada and its packages allow the abstraction of classes of objects and the actions to be taken on them, such as complex numbers that can be defined by package specifications with a separate body of code to implement them. The use of generics makes it possible to provide an even higher level of abstraction by providing a parameterized package specification to handle something like a list or an array; Ada generics will "instantiate" those specifications to allow the elements of the list or array to be a particular type.

Data abstraction is achieved in several ways. In the case of enumerated data types—which most people think were introduced in PASCAL but in reality had their origin in COBOL—it is allowable to mention a data object without worrying about its internal representation in the computer. This is the simplest form of abstraction. The more interesting kind of abstraction is *packages*, which are created to define and operate on data objects or types for other users while hiding the details of the implementation. Within the package specification it is even possible to hide part of the descriptions.

Generics appeared in a primitive form in PL/I but the formal separation of specification and implementation seems to be unique in Ada.

Information Hiding and Encapsulation. These concepts are related but distinct. *Encapsulation* refers to the ability to group information in a modular fashion that prevents access to the insides of modules except in very specific ways. *Information hiding* means that the programmer is unable to use "hidden" data except in very specific regulated ways.

Encapsulation is achieved by using functions and procedures in Ada (which are certainly not new), by packages, and by separate compilation. The latter facility in Ada is more powerful than anything existing before; the compiler combines data and procedure names properly without the user having to worry. In FORTRAN COMMON, and JOVIAL COMPOOL, the user must declare that the variables are going to be used in many different ways.

Information hiding is achieved via packages and tasks as well as "private types"; only the information provided in the specifications is available for use by other parts of the program and the executable body is not available. Even the specifications can be further restricted so that only a few operations on the data are allowed by users of the package.

Separation of Specification from Implementation. This is achieved by the Ada packages. The specifications can actually be compiled separately.

Reusability. Ever since the second square root routine was written, the programming field has lost adequate control of reusability. Although just writing Ada programs will not make them reusable automatically, the facilities in Ada make it easier to do so than any other language. The basic features that support reusability are the packages, generics, and data abstractions aided by various portability features. As with any aspect of reusability, care must be taken initially to ensure that quality is achieved.

Separation of Logical from Physical Concerns. From the first significant programs written in a high-level language—namely FORTRAN—there has been concern about the relation between the logic expressed by the program and the physical machine (and its compiler) on which the program was to be compiled and run. Although Ada has portability as a major design goal, it was also clearly recognized that in some cases portability prevented getting maximum efficiency from a program. In addition, the need to handle special machine interrupts, and nonstandard I/O (both of which are important in embedded computer systems) led to the feature known as "representation clauses," which allows the programmer to indicate specific machine characteristics associated with data. Programs can also provide compiler directives, which involve aspects of computer hardware such as storage.

Features of this kind previously appeared in the JOVIAL COMPOOL and the COBOL Data Division.

Portability. One of the major techniques by which portability can be achieved is allowing the user to control numerical data via data typing, and by specifying the precision, rather than indicating the number of bits or bytes. This concept was introduced in PL/I and carried over to an even greater extent in Ada. The facility for the programmer to specify the ranges on integers also assists portability by reducing concerns about word size. Naturally portability is aided by data abstraction, and by the common aspects of the programming support environment. Generics assist because they make it unnecessary for the programmer to define specific data types for each case.

Modularity Features. Several of the Ada features help achieve modularity and a few of these appear to be unique. A library that can be accessed from the program has not been formally available in most *languages*, although it was assumed to be available in COBOL and certainly has been provided by some compilers and/or operating systems. Procedures and functions are common to almost every language. Modularity is enhanced in Ada chiefly by packages, generics, and by separate compilation units.

Readability. Ada is more readable in large segments than other block structured programming languages because the package specifications and generics enable the reader to see the general structure of the program more easily. Strong data typing makes it easier to discern exactly what is being said in the program. Ada does not have the English language naturalness of COBOL, but on the other hand it does not have the cryptic notation of APL. In the final analysis—as with any language—readability depends heavily on how well the writer organizes the material (e.g., naming conventions, comments), and how well the actual code is written.

Verification. Verification is the software engineering concept that Ada does not formally support, in the sense that Ada does not have specific built-in verification-motivated facilities such as assertions. On the other hand, the compiler can find errors of many kinds either at compile time or at object time.

If one contemplates a greater level of abstraction, note that one purpose of verification is to help achieve reliability. That is, verification is not the end in itself but rather the means of achieving the desired goal of reliable software. Reliability itself is definitely aided in Ada by strong data typing, by separating the package specification from its implementation, and separating the few physical descriptions from their logical characteristics. The exception mechanism in Ada also aids reliability.

SUMMARY AND CONCLUSIONS

There has been frequent reference to an Ada "program" meaning (1) the method by which the language was developed, and (2) all the other related activities that have been underway since then. It is essential to understand that the Ada language is not the same thing as the Ada "program" when "program" is used in this context. It is my firm personal belief that the value of the Ada "program" is almost independent of specific language characteristics. In other words, the uniqueness of Ada as a language exists in large part because of the "program" within which it was developed. In my personal judgment, even if Ada was not as good a language as it is, it would still be "*not just another* programming language," at least in this broad sense.

One of the most unusual aspects of the entire Ada effort has been the unique blend of a number of inherently opposite concepts, specifically:

- Competition and cooperation: some of the language designers whose early work was eliminated went on to participate in the later language design work of their previous competitors.
- Committee design and a single designer: the value of access to a large group of knowledgeable people was maximized; the lack of cohesiveness that frequently comes from committee votes on technical matters was avoided by having a single designer with the authority to make final decisions.
- Military and nonmilitary involvement: although the effort was sponsored by the Department of Defense, there was significant nonmilitary, as well as non-U.S., involvement.

- The importance of both technical and nontechnical issues: almost every language that has been created in the past has been judged and considered solely on its technical merits.

Ada is a unique combination of specific technical features such as packages, tasking, generics, strong typing, etc., and no language that has preceded it has contained *all* of these features in a neat technically integrated fashion.

It is important to understand that being unique does not automatically mean that something is good. Furthermore, an activity that is unique is not necessarily successful, although I personally believe that Ada is a good language that will in fact succeed. But of course, one must decide what it means to have Ada succeed. In my judgment, this means three major objectives must be achieved: (1) it will be useful in embedded computer systems (which was its major objective); (2) it will significantly reduce Department of Defense life-cycle software costs (and presumably the cost for any other organization that chooses to use Ada); (3) it must have a significant impact on the computing community outside the Department of Defense.

Ada is here to stay; Ada is *not* a passing fad; Ada is *not "just another* programming language."

Acknowledgments. I am grateful to the following people who made constructive suggestions on an early draft of this paper: Harley Cloud, Larry Druffel, Anthony Gargaro, Jan Lee, John Rymer, and Doug Waugh.

REFERENCES

References 2, 4, 5, 6, 13, 19, 20, 21, & 25 are not cited in text.

1. *ACM Ada Letters* (continuing), including many bibliographies.
2. ACM AdaTEC. *Proceedings of the AdaTEC Conference on Ada.* (Arlington, Va., Oct. 6–8, 1982), ACM, 1982.
3. ANSI/MIL-STD-1815A-1983. *Reference Manual for the Ada Programming Language.* American National Standards Institute, Inc., 1983.
4. Barnes, J.G.P. *Programming in Ada.* 2d ed., Addison-Wesley, Reading, Mass., 1984.
5. Barnes, J.G.P., and Fisher, G.A., Jr., Eds. Ada in use. *Proceedings of the Ada International Conference.* (Paris, France, May 14–16, 1985), Cambridge University Press, Cambridge, England, 1985. Also available as *ACM Ada Letters V,* 2 (Sept.–Oct. 1985).
6. Booch, G. *Software Engineering with Ada.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, Calif., 1983.
7. Carlson, W.E. et al. Introducing Ada. In the *Proceedings of ACM Annual Conference.* (New York, Oct. 1980), ACM, 263–271.
8. Crafts, R. Commercial applications software in Ada: A reality. *ACM Ada Letters I,* 4 (May–June 1982), 46–54.
9. Department of Defense. Requirement for High Order Computer Programming Languages, "STEELMAN." June 1978.
10. Department of Defense. Preliminary Ada Reference Manual and Rationale. *ACM SIGPLAN Not. 14,* 6, Parts A and B (June 1979).
11. Department of Defense. Requirements for Ada Programming Support Environments—STONEMAN. Feb. 1980.
12. Department of Defense. Military Standard Common APSE Interface Set (CAIS), Proposed MIL-STD-CAIS. Jan. 31, 1985.
13. Druffel, L.E. The potential effect of Ada on software engineering in the 1980's. *ACM Softw. Eng. Not. 7,* 3 (July 1982), 5–11.
14. Druffel, L.E., and Buxton, J.N. Requirements for an Ada programming support environment: Rationale for STONEMAN. In *Proceedings of COMPSAC 80.* IEEE Computer Society, (Oct. 1980), 66–72.
15. Druffel, L., Redwine, S.T., and Riddle, W.E. The STARS program: Overview and rationale. *Computer 16,* 11 (Nov. 1983), 21–29.
16. Fisher, D.A. DoD's common programming language effort. *IEEE Computer 11,* 3 (Mar. 1978), 24–33.
17. Goodenough, J. The Ada compiler validation capability. In *ACM SIGPLAN Not., Proceedings ACM-SIGPLAN Symposium on the Ada Programming Language.* (Boston, Mass., Dec. 9–11, 1980), *15,* 11 (Nov. 1980), 1–8.
18. Ichbiah, J.D. Ada: Past, present, future—An interview. *Commun. ACM 27,* 10 (Oct. 1984), 990–997.
19. *IEEE-CS Computer 14,* 6 (June 1981).
20. LeBlanc, R.J., and Goda, J.J. Ada and software development support: A new concept in language design. *IEEE Computer 15,* 5 (May 1982), 75–82.
21. Ledgard, H.F., and Singer, A. Scaling down Ada (Or towards a standard Ada subset). *Commun. ACM 25,* 2 (Feb. 1982), 121–125.
22. Lieblein, E. The DoD software initiative—A status report. *Commun. ACM 29,* 8 (Aug. 1986), 734–744.
23. Wegner, P. Self-assessment procedure VIII (Ada). *Commun. ACM 24,* 10 (Oct. 1981), 647–678.
24. Whitaker, W.A. The U.S. Department of Defense common high order language effort. *ACM SIGPLAN Not. 13,* 2 (Feb. 1977), 19–29.
25. Wichmann, B.A. Is Ada too big? A designer answers the critics. *Commun. ACM 27,* 2 (Feb. 1984), 98–103.

GUIDE TO REFERENCES

General History and Background: [7], [16], [24]
Introductory Language Material: [4], [6], [20], [23]
Detailed Language Specifications and Requirements: [3], [9], [10]
Environment: [11], [12], [14]
General Material (many topics): [1], [2], [5], [19], [22]
Other: [8], [13], [15], [17], [18], [21], [25]

CR Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*validation;* D.2.6 [**Software Engineering**]: Programming Environments; D.2.m [**Software Engineering**]: Miscellaneous; D.3.2 [**Programming Languages**]: Language Classifications—*Ada;* D.3.3 [**Programming Languages**]: Language Constructs; [**K.2 History of Computing**]: Ada; [**K.2 History of Computing**]: Software
General Terms: Design, Languages, Management, Standardization, Verification
Additional Key Words and Phrases: programming language comparisons, programming language history

Author's Present Address: Jean E. Sammet, IBM Federal Systems Division, 6600 Rockledge Drive, Bethesda, Maryland 20817.