ADA: PAST, PRESENT, FUTURE An Interview with JEAN ICHBIAH, the Principal Designer of Ada

Since the mid-1970s, the U.S. Department of Defense has been laying the groundwork for a major new computer language that may substantially displace FORTRAN and COBOL in the years ahead. With Ada just now starting to be used, Communications *felt it would be timely to conduct an interview with Jean Ichbiah, the principal designer of this new language. Ichbiah discusses the evolution of Ada, evaluates its success so far, and speculates on its future.*

Q. **How did the Department of Defense (DoD) come to sponsor a new computer language?**
Ichbiah. This is not the first time the DoD has embarked on trying to standardize a new language. They did that successfully with COBOL. What made COBOL a widely used language was not only its design merits. Rather, it was that the DoD required COBOL to be used on all defense contracts. Even back in the early 1960s, the DoD had a philosophy of trying to bring some order into software, to permit more portability of programs between different computers, and thereby to enhance productivity.

I mention COBOL because it set an important precedent. COBOL is a successful instance of a language being spread to a worldwide community. If you compare the history of COBOL to that of PL/1, you'll see that even IBM didn't have enough weight to spread a language; but the DoD did.

Q. **What were the DoD's motives in pushing COBOL back in 1960?**
Ichbiah. The motives then were exactly the same as they are now: to stop the proliferation of languages and dialects. At that time most programs were written in assembly language; they were difficult to maintain and therefore costly and unreliable.

COBOL—an acronym for "common business-oriented language"—was an attempt to move toward higher programmer productivity and better interchangeability of programs between different machines.

Now, coming to the genesis of Ada: Around 1975 the DoD did some studies to see what could be done to control escalation of software costs. At that time, they were spending over three billion dollars a year on software.

Q. **Was that for developing new software, for maintaining existing software, or both?**
Ichbiah. That was the total software budget for developing new software and maintaining old software. When the DoD first realized the magnitude of this amount, they started to analyze the situation and discovered that they were using about 400 different languages and dialects—quite a large number.

The first major conclusion the DoD derived from these studies was that to control software costs, they would have to make a major investment in tools to enhance productivity. The second major conclusion was that they would need to improve the average programmer's level of programming methodology. The development of the new Ada language was itself a consequence of these first two conclusions: It was out of the question to develop tools and education strategies around 400 incompatible languages, few of which had been formally designed.

Q. How can you explain such a proliferation of dialects?
Ichbiah. Well, there would be an application, and some clever system programmer would create a small change in the current language to make it a little "easier" for that application, thinking he could improve productivity. What he did not realize was that for the next 20 years, 10 generations of programmers would have to learn that dialect in order to maintain the program.

Now, think of a programmer maintaining this program. It looks like COBOL, so he thinks it is COBOL. It's going to take him some time to realize that it does not mean the same thing. This new dialect has imposed an enormous learning burden, which is amplified by the fact that it's hard to develop training methods and effective program documentation tools for a dialect that is not widely used.

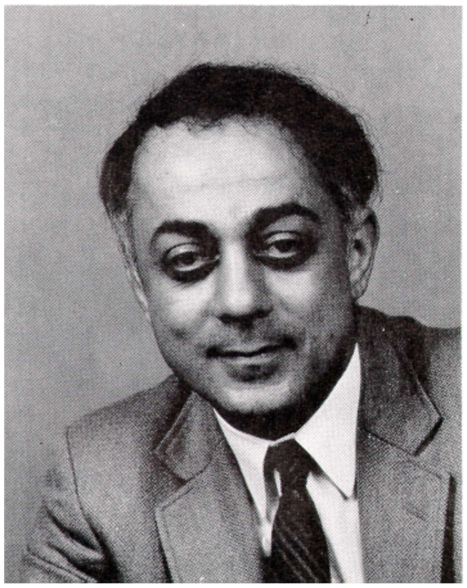
Spreading a language is costly. It requires having a very rigorous language definition to ensure that people who communicate with this language mean the same thing when they use it. From my experience with Ada, I can tell you that the development of a rigorous standard requires a significant amount of work. The Ada development started in 1977. We had a proposed standard document in July 1980. It took from July 1980 to February 1983 to produce the ANSI standard for Ada. In producing this standard, my group had to review 7000 comments produced by experts from 15 different countries. This involved hundreds of authorities around the world. It is of course out of the question to do that for 400 languages. When you have that many languages, they are not going to be as rigorously defined.

Training is also a factor. If you look in technical bookshops, you will already find more than 50 different books on Ada. If you want to train programmers on Ada, you already have good material to teach them the language.

Q. Is lack of portability another liability of language proliferation?
Ichbiah. The portability issue has several dimensions: portability of programmers, portability of programs, and portability of expertise.

In his Turing lecture three years ago [3], Tony Hoare expressed some doubts about working toward a standardized single language. Another alternative, he suggested, was to start from a smaller language, like Pascal. Then whenever you had certain applications, you could make specialized extensions of the language toward these applications.

Many of us consider this approach to be dangerous. It fails to address the pitfalls of multiple languages and dialects. People considering a given domain find that



The Ada language was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a concern for programming as a human activity, and efficiency. As head of the Ada design team, Jean Ichbiah worked to synthesize these goals into a viable programming language. This effort culminated with the achievement of ANSI standardization in 1983.

Before working on Ada, Ichbiah headed the Programming Research division at Cii Honeywell Bull, where he supervised research in the design and implementation of programming languages and projects in the areas of programming methodology, software tools, and compiler optimization techniques.

Ichbiah has authored numerous publications in the international literature on programming languages, compiler technology, software productivity, and programming methodology, and is the founder and chairman of Alsys, Inc., a company that develops and markets Ada-related products and services.

they need extension A for a certain application. Another group in a different domain develops extension B to meet another application. So far so good. If A and B were in totally different worlds, we could perhaps survive with a situation like this.

The difficulty arises when these applications are so successful that we want to mix them, put the two together on a computer and have them interact. This puts us in a very dangerous area because of the language extensions. Although A and B look very much alike, they are sufficiently different to create the risk of adverse consequences. When the first group wrote something, they meant something very specific. People from the second group will place their own—different—interpretation on what the first group wrote. As a result, the program will execute something that is not what was intended.

In developing a unique language, we want a language for which there is a single interpretation for all people who use it.

***Q.* Who defined Ada's requirements?**
Ichbiah. The requirements were developed by the DoD under the leadership of David Fisher.[1] He consulted experts from the military, from industry, and from academia, not only in the United States but worldwide. It is the first time that a completely separate group has defined the requirements for a programming language.

If you were planning a house, you would have your dreams and your requirements; you would have so many children and need so many rooms. *You* would specify your requirements—not an architect. You would then select an architect and say, "These are my requirements. Build me a house that satisfies them." Of course there is interaction between you and the architect. But it is important that these two functions—the specification of the requirements and the design that satisfies them—be separate.

If you look at the design of previous computer languages, it didn't happen that way. Take Pascal, for example. The same person decided what the language should do and how it should carry out those objectives. It's like having an architect design your house to suit his needs. It is a very good sign of the maturity of our profession that there was a completely separate group defining Ada's requirements.

I see the global architecture of Ada as a cathedral, with all the architectural lines interwoven in a harmonious manner. I would not do it differently if I had to do it over again.

The development of the language requirements took nearly three years. The Strawman requirements appeared in 1975; Woodenman also appeared in 1975, Tinman in 1976, and Ironman in 1977. These were documents stating, in progressively greater detail, what requirements the language was expected to satisfy.

The Ironman requirements attracted me very early in the game. The preamble demonstrated a keen understanding of today's software problems. The stated key objectives were to improve the reliability, readability, and maintainability of programs, along with more classical goals like improving portability and efficiency.

If these requirements had been written in 1960, they would probably have emphasized portability and efficiency, but I doubt if they would have stressed reliability and maintainability above all. These concerns came to the fore in the 1970s.

I found it interesting that the Ironman document stated, "The language should promote ease of maintenance. It should emphasize program readability over writability. That is, it should emphasize the clarity, understandability, and modifiability of programs over programming ease" [4]. That was quite new. For many years the emphasis had been on creating languages that made it possible to write programs very fast. It took years for the software industry to realize that this was an improper goal: The development of a large program may take less than two years, whereas that same program may have to be maintained for over 20 years. Even if it takes a little longer to write it, it is far more important that a program be readable than writable. For 20 years after the program is written, programmers will have to read it to maintain it. And the easier a program is to read, the easier it will be to maintain.

***Q.* How did you happen to get involved in Ada?**
Ichbiah. I have been working on programming languages most of my life. In 1972, I was working for Cii Honeywell Bull and was given the task of developing a system implementation language. We called it LIS, which in French stands for "Langage d'Implémentation de Systèmes."

I was given objectives quite similar to what would later be the Ada requirements. The language had to improve the reliability of producing operating systems, and it had to improve maintainability. It was far more important that the programs be readable than easy to write. We produced a first version of LIS in 1972 and the final version around 1974.

During all these years, I participated in the work of international groups that were searching for better languages: the IFIP working group WG 2.4 on System Implementation Languages and the Purdue-Europe "Long Term Procedural Language" group. Within these groups there was a general feeling that the major languages in use (COBOL, FORTRAN, PL/1) were really outdated. These might have been good languages, but they reflected the technology of the early 1960s; they suited the economic ratios (i.e., hardware costs versus programming costs) of a time dominated by the overwhelming price of hardware. All the effort was directed at making sure that the hardware would be operating 24 hours a day. Today, hardware is cheap, and we want to use it to maximize the productivity of programmers. The relationship is inverted.

So there was this feeling that we had to develop a better language technology. In the United States, Colonel William Whitaker was starting to organize what would later become the Ada program. I met him in 1975 in one of these working groups, and he showed me the Woodenman and Tinman requirements. I found them extremely interesting, and we started to interact. I showed him the kind of work we had done on LIS, which was a first step in the direction DoD was looking for.

In 1977, Whitaker encouraged us to bid on the Ada language design project. We were one of 20 bidders.

[1] See Grady Booch's *Software Engineering with Ada* [1] for a more detailed account of the history of Ada's development.

From these, the DoD selected four teams. They were color-coded Green, Red, Blue, and Yellow. We were the Green team. That is why the covers of the Ada manuals are always green—for a long time it was called "the Green language."

The DoD wanted the four teams to develop sketches of what the language could be. After six months there was a public evaluation of these four sketches. In 1978, the Green and Red teams were selected to continue, and in May 1979 the final choice was announced: The Green language became preliminary Ada.

After 1979, my Green team continued alone during the so-called "Test and Evaluation" phase, during which about 100 teams around the world tried to recode applications in this preliminary Ada language. These teams reported the drawbacks they found, what had to be improved, etc. Using this first experience and knowledge, we produced, in 1980, a proposed standard for the new language and finally, in February 1983, the Ada ANSI standard.

Thus, it took from 1977 to 1983 to produce a standardized language. The process involved more than 1000 people from around the world.

Q. How could you handle the comments from such a large number of people?
Ichbiah. Clearly, this was a problem in itself. During the ANSI standardization, my team reviewed and processed around 7000 comments on the proposed standard and on later drafts. The nightmare I always had was that somebody would show me a flaw in the language definition, that I would agree that there was a defect, but that I would forget to correct it. This was possible when there are 7000 comments. If you don't make a change on a comment you disagree with, that's fine. That is the designer's prerogative. But if you agree with a suggested change and forget to make it, it's terrible.

So we computerized all these comments, created a database of comments. When we read comments, we attached our own reactions to them. That enabled us to produce mechanical checklists, just to avoid this nightmare of forgetting to make a change that I had agreed was needed.

Use of such a database of comments is probably another thing that is new in language design. It is similar to what a building constructor uses to make sure he does not forget certain stages of the construction process.

Q. Have there been any major changes in Ada between 1978 and 1983 as a result of the feedback you received?
Ichbiah. Yes. But first let me say that I see myself really as an architect. My work was not to invent new things; it was not research work, it was architectural work. I had to integrate the best available materials to construct the building that would best suit the requirements of the users.

As with any design, you end up with a product that is not necessarily perfect, but you make the best possible compromises to satisfy the requirements.

Q. Looking at the design you had in 1979 when you first completed the Green language, how would you compare that with ANSI Ada?
Ichbiah. If you looked at Ada from a distance, you would not see much difference between the initial language and the final language. The structural lines, the foundations, were correct to start with. If they had not been correct, no further improvement would have been possible. So the structure has remained stable. But public feedback has resulted in a very high level of polishing, which has made Ada more usable to programmers.

Q. There have been critics, such as Tony Hoare, who consider Ada to be too complex. Would you care to comment on that?
Ichbiah. Some of these critics take a mathematical view of complexity that is not necessarily applicable: They usually underestimate the abilities of the human mind. My own view of complexity and simplicity is more influenced by architecture. (In fact, my training initially was as a civil engineer.)

The human mind has an incredible ability to understand structures. Provided it understands the major lines of a structure, it will make inferences and immediately see the consequences. When you judge something, the complexity is not in the details but in whether or not it is easy to infer details from major structural lines.

From this point of view, I consider Ada to be very simple. The experience we have had in training people in Ada is that once they understand the major structural lines—something we are usually able to convey in a three-day course—they find that they can rely on their intuition to supply the detail correctly.

Consider a large auditorium. There are two ways to look at it: the designer's and the mathematician's. The designer creates a thick set of construction documents. If you gave these documents to a mathematician, he would probably say, "These documents are terribly complex, and it is therefore risky to construct this auditorium. People will never find their way out . . . . I urge you to not construct that auditorium because it is going to collapse . . . ."

Now, the reason for preparing these construction documents is to make sure that the contractor will build exactly the auditorium that is planned, that the electric wiring will be where it belongs, that the curve of the seats will be such that everybody can see very well, that the acoustics will be ideal, that sprinklers and other fire control provisions will be effective, etc. The complexity is in the design.

Now, that does not mean that the user of the auditorium is going to find it complex. Walking into the room, he immediately understands its structure and finds his way to his seat directly.

This analogy applies very much to Ada. The major architectural lines of this language are simple, which means that users find their way very easily in using it.

Q. **Is Ada in its final form right now? Is it to be frozen as it now is for the next 30 years? Under what circumstances do you see changes being made?**
Ichbiah. Right now the language is certainly frozen. The ANSI standard was issued in February 1983, and the standard is a freezing point. About every five years or so one does a revision of a language. The only languages that are not revised are dead languages like Sanskrit or Latin. The needs of the computer profession are evolving. And as they evolve, languages need to be revised to serve their users better. Ada is no exception. Around 1990, the need for an Ada revision is likely to arise.

Q. **Would you expect that to be a relatively minor revision?**
Ichbiah. It is probably too early to say, but I tend to think it would be a very minor revision. Aside from revisions, though, there will always be a need to provide interpretations. This is true for every language. This is similar to what the French Academy does in standardizing the French language. Every now and then they meet to decide what to do about certain ambiguities. Definitions are made as precise as possible, but sooner or later somebody comes up with a point that needs to be resolved.

Q. **Is it not rather unusual to have a language standard so early in the development?**
Ichbiah. The French humorist Pierre Dac once said, "The Leaden Rule is: shoot first, aim second, think later." That is precisely what was done in developing previous computer languages. People designed a language. Then, when they had many compilers, they discovered that different compilers did not implement the language consistently. Only then did they develop a standard. That is doing things back to front. By the time you have your compilers, the harm is already done and you can only standardize on some more or less common sublanguage.

Ada represents the first instance in the history of programming languages of things being done in the right order. That is an important sign of the maturing of this profession.

For Ada, we first created the design—the major architectural lines of the language. Next, we standardized the language, made sure that the description of the language was precise, and that everybody agreed on the definitions. Then a validation facility was produced: a tool to check that compilers conform to the standard. Finally, the compilers appeared.

For users it is important that the compilers interpret the language consistently and in the same way. It is their best guarantee of program portability. The validation actually checks that the compilers implement the full language, and not just a subset. And to a certain degree, it checks that there is no superset. Both of these aspects are essential. We want the language to be the same for all users. That is their best guarantee that an Ada program will be usable on all computers.

Q. **Looking back, what do you think are the key factors that enabled your team to win in the Ada competition?**
Ichbiah. A key reason is that I started this work five years ahead of my competitors— I am referring to my work on the LIS language at Cii Honeywell Bull. We had had experience in developing a language with requirements that were very similar to Ada's.

It turned out that one of the superiorities people found in our Green language (Ada) was its aesthetic dimension. Perhaps my competitors came from a more theoretical and mathematical background, and less from an engineering background, as I did.

The designers of the Red language, Ben Brosgol and later John Nestor, took a more mathematical approach. Some of their ideas were actually so good that I borrowed them in a later phase. They devised their language to appeal to mathematically oriented people. But the Red language did not have the kind of aesthetic dimension that the Green language had. The Green language appealed to people. They liked it. They were pleased with it. This appeal came from what I mentioned earlier: that the major program structure was clean and apparent.

Ada's *package structure* also had an immediate appeal. This concept—probably the key feature of the language—creates a very clean separation between the *visible part* (the interface with the user) and the *package body* (the domain of the implementer). The simplicity of Ada's textual structure lets people see immediately how they can write programs that will be appealing to read.

Q. **You said that a key feature of Ada is its package structure. What is that, exactly?**
Ichbiah. Let me use an analogy: Consider a watch. We can view it as an Ada package. It has a set of entry points: a procedure (or button) called *Select* to select a given function of the watch; a procedure called *Set*, to reset the date; and a procedure for displaying the current time. This is the user's view of the watch.

If there was nothing more to the watch than these buttons, it would not be able to perform its functions. There has to be something inside the watch: a mechanism to *implement* all the promises on the outside. In Ada, you can collect the facilities you want to provide into a package. The user's view of these facilities is defined by a *package specification*; it is the *visible part* of the package. The programs that provide their mechanism are what we call the *package body*. It is a general rule of Ada that the specification (the user's view) and the implementation are always presented separately.

This package structure had immediate appeal. You can readily see how this concept applies in everyday programming. You have a programming team with an application. The team agrees on the different packages that they need to provide and on the users' interface to each package. Each subteam goes away and develops the package bodies—the means for implementing the promises.

Q. Do you think that this package concept was the main reason the DoD selected the Green language?
Ichbiah. There were other reasons, too: For example, another principle extensively used in Ada is *linear reading*. How do you read regular text? You read the first line first. When you're finished with that, you read the second line. When you reach line four, your knowledge comes from lines one through four. You don't know anything about the lines after that.

Now, in the past, many programming languages didn't conform to this linear-reading property. Ada does. You read an Ada program as you read a normal text, line by line. When your understanding has reached a given line, you don't depend on things that come later. This approach is really an extrapolation of an important idea introduced years ago by Dijkstra. In 1968, he wrote a famous letter in which he stated: "Our mind is better geared to understanding static structures than dynamic ones" [2].

Programs are only going to be correct if we *understand* that they are correct. If we can develop programs in such a way that we can statically understand whether they are correct, they are more likely to be correct. In Ada we tried to reduce the dynamic dimension of program execution. Whenever we can write our program so that we can qualify program points by static properties, we make it much easier to understand, and we tremendously improve our chances of writing the program correctly.

When we construct a building, we are not trying to create the *ideal* shape. Ideal in terms of what? We are trying to create a shape that we can construct. So we are always influenced by constructability factors.

It is the same in writing a program. We are trying to write a correct program. We could construct a very intricate, complex program, but we might never finish the job or get it to do what we want. So we are limiting ourselves to programs that we can understand.

Q. Does program readability depend on how the programmer writes his program or on the language he chooses to write his program in?
Ichbiah. It depends on both. Of course it is possible to write programs that are difficult to read in any language. But a language like Ada allows you to write more readable programs than languages like FORTRAN or COBOL do. Certainly, writing a readable program demands a special effort from the programmer. But with Ada, the programmer has a powerful tool that enables him to write good programs.

Q. Could you talk about where Ada is today? Is it being used by the DoD right now? Is it being used in industry?
Ichbiah. Ada exists today—there are compilers that compile Ada. As in other domains, when a new technology appears, initial interest is high—this was the case when Ada appeared in 1980. Then there is a time when professional skeptics take over and say, "The technology is too complex, they'll never get it standardized. We'll never understand it. They will never be able to implement it." I think we are beyond that stage now that we have Ada compilers.

Q. Who came out with the first compiler?
Ichbiah. The first compiler was developed at New York University by Robert Dewar and his group. Its significance is more of a historical nature, because it was an interpreter—rather slow, and more of a teaching tool than anything else.

The second compiler, developed by Rolm and Data General, has more industrial importance. Validated in June 1983, this was developed for a Data General Eclipse minicomputer. The third compiler to be validated was created by Western Digital for their MicroEngine; it was the first compiler for a microcomputer.

Several companies around the world are producing compilers for many different machines. In July 1984 Digital Equipment announced prevalidation of the VAX compiler. My own company, Alsys, is producing Ada compilers for microcomputers.[2]

Q. Are there any mainframe compilers yet?
Ichbiah. Not yet. But there are several companies working on it. For example, there are two major government-funded projects in the United States, one by Softech, the other by Intermetrics. They are creating compilers for machines like the DEC VAX and the IBM 370. By the end of 1984, I would not be surprised if there were 20 validated compilers, including some for mainframes. Some compilers are being developed under DoD contract. Others are being developed by private sector companies—computer manufacturers, software houses, etc.—because they see a market.

Q. Is the DoD using Ada right now?
Ichbiah. The DoD has issued an important directive stating that all mission-critical applications must be programmed in Ada as of January 1984. They must have been waiting until they had actual evidence of the feasibility of Ada compilers. This evidence was brilliantly provided by the Data General compiler several months ago. This new DoD directive is definitely a move by the DoD to do something similar to what they did for COBOL in the 1960s.

Q. Is Ada going to be used within the DoD just to write new programs, or do you also expect to see people rewriting existing programs in Ada?
Ichbiah. I think they would tend to use Ada for new programs at first. There may come a time when the proportion of new programs is so great that it will be worthwhile to rewrite the old ones. Initially, there will probably be very little rewriting of old programs.

[2] This was the compiler situation at the time of this interview, in early 1984—*Ed.*

Q. To what extent are industry and business going to use Ada—especially those industries not connected with the defense establishment?
Ichbiah. The only thing that will stop the private sector from using Ada is not having Ada compilers. The minute compilers are widely available, Ada will provide industry with solutions far superior to what they have had in the past. Ada will be in great demand in industry. The choice will be more and more for Ada in the private sector. My own company is actually focusing 100 percent on the private sector. We consider the defense sector a subfield of the general sector. After all, COBOL was very strongly supported by the DoD, but who would say today that COBOL is a military language?

Q. Suppose I were the president of a company. Why would I be interested in Ada? What would be the bottom-line benefit for my firm?
Ichbiah. The benefits are in several dimensions. You are interested in programming productivity, reliable programs, easy maintainability. This means you are interested in having programs of sufficient readability that you will be able to assign new programmers to maintain these programs. And you don't want it to take years before they can understand what those programs mean.

You are interested in efficient programs, although perhaps to a lesser degree than in the past. Nevertheless, Ada provides that too.

You are also interested in being able to move people quickly on the job—portability of programmers. If you hire a new programmer and your company is using Pingol 3 (or whatever), you'll have a problem teaching him Pingol 3. Finally, Ada will provide you with a better answer for reliable programming.

Q. Will Ada substantially reduce programming cost, that is, the cost of writing an initial program?
Ichbiah. Ada will reduce the *life-cycle* costs. I don't think it will necessarily reduce initial programming costs—it may even increase them. But what good does it do to reduce the cost of programming if you spend more time (money) for debugging, servicing, and maintaining your program? Ada looks at the total life-cycle cost. It is there that you produce the real gains.

In programming in Ada, people have to put more of their intention down than they would in, say, FORTRAN. They have to think a little harder about what they are trying to do. Consequently, they may spend more time writing the initial program. But this program is much more likely to be correct. They are more likely to spend much less time debugging and maintaining that program later on.

Q. Will Ada programs still have to be debugged?
Ichbiah. Yes, but it will require less time than before. More errors will be picked up by the compiler. Maintaining the programs will be much simpler because the programs will be more readable.

Q. Would an industrial company use Ada to address both engineering and business problems?
Ichbiah. Yes. A firm would use Ada both for engineering and for business problems, including real-time systems, which is another dimension that you could not easily treat with previous languages.

Q. Is Ada something all businesses and industries should be interested in?
Ichbiah. Ada is still a traditional language in that it is efficiency oriented. At the other end of the spectrum, there are applications where you might not care if an application runs a thousand times slower. Ada is still meant for programming applications where efficiency is important. If you are in a domain of applications where you don't care about efficiency at all, there may be other tools.

Q. Could you highlight Ada activity in Europe? What countries and industries are particularly interested?
Ichbiah. There is tremendous interest in Ada in Europe. There is an organization called Ada-Europe that is very similar to ACM's AdaTEC. There is a lot of interest in France, England, Germany, Italy, Denmark, and Sweden. In Sweden the telephone industry has shown very keen interest. At seminars I give in France, many attendees are people from the data processing departments of banks and insurance companies. They are trying to determine what place Ada should have in their three-year plans.

Q. Is there more interest from data processing people than from scientific and engineering people?
Ichbiah. There is interest from engineers, but I was expecting that type of interest. I was surprised by the early interest from data processing people. I was expecting that interest to develop later.

Q. Ada statements are basically in English. When Ada is used in France, Sweden, etc., do they use a version with words in their native language?
Ichbiah. There are only 63 key words in Ada. Even though these are English words, they are not translated. They don't bother a Swede or a Frenchman, to whom they are just symbols.

Q. Do you think Ada will be around forever, or do you foresee that Ada will start to be displaced by another language in 10 or 20 years?
Ichbiah. I expect that Ada will eventually be displaced by other means of formulating problems on a computer. Technological evolution will alter the economic factors (i.e., hardware versus software costs). Ada corresponds to our time because it is still a very efficient language. Thirty years from now, though, we may be able to produce machines that are a million times faster at the same cost. Obviously things that we consider economically unjustified now might become justified then. In that case, efficiency-oriented lan-

guages like Ada may become obsolete for more and more tasks.

I certainly hope that 30 years from now our view of efficiency will be radically different. In 1984, any programmer has a certain intuition about the amount of computer resources he is willing to devote to the achievement of a certain task. It is an economic judgment. Ultimately, it translates into dollars spent for a given task.

Let me give an extreme comparison. In solving a problem in 1984, let us assume I have a choice between doing an exhaustive search that would mean evaluating 10,000 solutions, and developing an elaborate algorithm that will find the best solution directly. How do I choose between the two? If the programming of the elaborate algorithm will take me two days, then this has a price.

Yet I may or may not be willing to pay that price. Let us suppose that the brute force evaluation of 10,000 solutions takes only one second of computer time. Then that is what I am going to choose. If, on the other hand, the brute force evaluation takes a year of computer time, I will invest the two days in programming.

When I use the words "brute force," I am implying it is a bad solution. But such an implication is not correct. Whether it is a bad solution depends on our perception of how much computer time we are willing to spend.

To summarize, Ada is part of a generation of languages that describe how to do things in an efficiency-oriented way—I am talking about efficiency in the sense of the amount of computer time required to execute a certain operation. When the economic factors are radically changed, we may need to use other tools.

Q. Is it important that a professional programmer today learn Ada? What is your advice to programmers?

Ichbiah. They should be aware of Ada. They should be prepared to learn its concepts. When programming in Ada becomes a possibility for them, they should switch if they think it is worth it. To reach this decision, they should learn the language and form their own impressions of Ada's potential. They should switch only when they are completely convinced that it is to their advantage to do so.

Q. It is easy for programmers to learn Ada? How long does it take a typical programmer to become proficient?

Ichbiah. Learning means going from a given state of knowledge to another state. So learning time depends not only on the ending point but on the starting point. Suppose you asked me how long it takes to learn English? I would answer that it depends first on whether you speak any related languages like French or German. It also depends on whether you are trying to reach the first level, whether you want to be a professional writer, and so on.

People who have already learned a structured programming language like Pascal are probably going to learn Ada in a very short time. For a Pascal programmer to reach an equivalent level of programming in Ada will take less than a week. And learning to exploit Ada's better techniques may only be a matter of a few additional weeks. Similarly, it will not take long for a FORTRAN programmer to reach the corresponding level of programming in Ada. Already at this stage he can reap certain benefits in readability and reliability.

I have taught Ada and LIS to people with very different backgrounds. I found that people get very enthusiastic. They are seduced by the cleanliness of its structures, and once that happens, they easily find their way in it.

Q. Are you personally satisfied with Ada as it is now?

Ichbiah. Now that we have completed Ada, I must say I am extremely pleased. I am pleased with the global architecture of the language. I see it as a cathedral, with all the architectural lines interwoven in a harmonious manner. I would not do it differently if I had to do it over again.

Q. What are the major lessons you learned from developing Ada?

Ichbiah. I have learned that when a team is motivated it can achieve fantastic things. I have learned it is possible to do a development like Ada with a large number of people around the world if you use network technology for communication. We used the Arpanet and Tymnet to communicate all the design documents. I have learned that it is not possible to create a language that satisfies ambitious needs if you are just one or two people sitting in a corner. We benefited greatly from the large number of international interactions that we had. But of course, a design like this has to be done with a single strong leader, since it is very important that the major architectural lines of a language be kept consistent: Consistency can only be achieved with one person defining the major lines.

REFERENCES
1. Booch, G. *Software Engineering with Ada.* Benjamin Cummings, Menlo Park, Calif., 1982.
2. Dijkstra, E.W. Go to statement considered harmful. *Commun. ACM 11*, 3 (Mar. 1968), 147–148.
3. Hoare, C.A.R. The emperor's old clothes, 1980 Turing Award Lecture. *Commun. ACM 24*, 2 (Feb. 1981), 75–83.
4. Requirements for high-order programming languages. *IRONMAN.* United States Department of Defense, Washington, D.C., Jan. 1977.

CR Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General—*Ada*; K.2 [**Computing Milieux**]: History of Computing
General Terms: Design, Languages