Ada and Software Maintenance

Major Patricia K. Lawlis, USAF

Computer Science Department
Arizona State University
Tempe, Arizona 85287

Abstract

The Ada language was developed to answer problems which were seen as the root of the high cost of software within the DoD. Ada is not the perfect language, but it is the best currently available. And the Ada Programming Support Environment (APSE) makes it more than just a language, but rather a technology capable of complete software engineering life-cycle support. All in all, Ada technology far surpasses the capabilities of any isolated language in providing for both effective system development and its subsequent maintenance.

Introduction

Over the years, software projects have had a reputation for being delivered late and over budget. At the same time, the cost of software maintenance on these projects subsequent to delivery has been found to be running 3 or 4 times the cost of development (by itself this is not necessarily bad, but on most systems it has made a bad situation worse). In the mid-1970s, the U. S. Department of Defense (DoD) recognized the need to do something to make software life-cycle costs more manageable. As a result, the DoD sponsored the development of a new computer language, eventually called Ada. The expectation was that in producing a common language which also supported the software engineering principles known at the time, that software quality could be improved at the same time as its overall costs could be reduced. A large portion of the expected cost savings was in the area of software maintenance.

This paper will explore the various life-cycle activities which have an impact on software maintenance, and discuss the effect that Ada can have on these activities. This discussion will include a look at the concept of the Ada Programming Support Environment (APSE) and the anticipated impact of the APSE on software development and maintenance. It is impossible to quote dollars and cents figures about the effects of Ada on maintenance until more systems have been developed in Ada and then have experienced some years of subsequent maintenance. However, it is possible to make some reasonable analyses about those aspects of Ada which can support significant changes in both maintenance practices and costs.

Section I is a brief historical background on the development of Ada and the APSE concept. Section II focuses on the management activities of software development which will have an impact on software maintenance, and Section III looks at technical development activities affecting maintenance. Each section also discusses how Ada's contributions in the development process can affect subsequent maintenance. Then Section IV summarizes the actual impact Ada should have on software maintenance activities.

I. Ada Technology

In the mid-1970s, when the Department of Defense (DoD) began looking for ways to stem the tide of skyrocketing software expenses, it started at the heart of the software development process by examining programming languages. However, through a series of calculated steps, the entire software engineering process was eventually addressed in this quest for reduced expenses.

The Ada language was developed to answer two specific problems which were seen as the root of the high cost of software within the DoD: the use of too many languages and the lack of support for software engineering principles in system development. The use of too many languages was solved by developing a language which supports all the features needed in DoD software and then standardizing the language definition to prevent differing versions. The Ada language contains numerous features made to specifically solve the second problem, supporting the known principles of software engineering.

The APSE concept was developed to solve a different level of problems than those addressed by the Ada language. It aims at the roots of productivity by providing both technical and management support for the entire life cycle of a project. In so doing, it directly addresses many issues of maintenance.

The Roots of the Problem

When the DoD began to examine programming languages, it had been using literally hundreds of them because of all the special purpose computer systems embedded in various weapons systems. Most of these "embedded systems" were programmed in assembly languages. Yet even those that were not used numerous versions of various "old" high order languages. It seemed that each DoD system was developed as a one-time special purpose system, and both the languages and the methods being used were "archaic" when measured in terms of technological advances. It would have been inconceivable to have continued to use vacuum tube technology for computer hardware, yet computer software development had not advanced from its state of the vacuum tube days. [5]

Because of the one-time nature of system development, few, if any, software tools were being developed to improve programming productivity. Software maintenance was hardly considered at all. Each new system development had neither the time nor the funds to spend much on developing tools up front. And tools from previous systems were specific to those systems, and thus could not be reused in the new development. Hence, programmers plugged away, using the little they had to work with, and continually "reinvented the wheel", over and over again. [5]

These problems were viewed as the roots of the "software crisis", and the DoD set out to do something to bring it under control. The natural direction seemed to be to look for a common programming language. With it, reusability could become a reality, and a tool base could be developed which would support known software engineering principles. This would not only result in fewer new software requirements but also in improved productivity in producing the new software that was still required. It would also improve the maintainability of the systems developed. [5]

The Search for a Common Language

The DoD began its search for a common language by comparing existing languages against a set of requirements for supporting embedded software. When no existing language was found to be suitable, the decision was made to develop a new one which would not only meet the requirements but would also embody the known principles of this newly emerging discipline of software engineering. After four contractors developed designs for the new language (each given a color as a code name), the green language proposed by Honeywell/Honeywell Bull was selected for further development. [5]

It was about this time that the language was also given the name Ada, in honor of Augusta Ada Byron, the Countess of Lovelace. Ada Lovelace (1815-1851), an associate of Charles Babbage, has been credited with being the world's first programmer because of her insights into how Babbage's machines might be programmed. [5]

By 1980 the Ada language design had been completed and standardized by the DoD, and in 1983 it was standardized by the American National Standards Institute (ANSI). In 1987 it was also recognized by the International Standards Organization (ISO). The language definition has been published in a document known as the Language Reference Manual (LRM), MIL-STD-1815A [11]. The DoD also copyrighted the name of Ada so it could not be used except for an implementation of the language that conforms to the standard defined in the LRM. An implementation which conforms must go through a validation process, and, as of 1 March 1988, 175 Ada compilation systems have passed this validation process [1]. The Ada Joint Program Office (AJPO) was established by the DoD to oversee the process of Ada technology insertion. [9, 5]

Language Features

A deep technical discussion of the features of Ada is beyond the scope of this paper. However, it is pertinent to examine the principles behind the language features which address maintenance concerns.

Ada is a procedural high level language. Its design was based on Pascal, so it contains structures similar in nature to those found in many of the popular procedural languages. It contains a number of additional mechanisms, however, which provide specific support for various aspects of software engineering. [5]

One of the important features provided by Ada is the ability to work with various levels of abstraction. This is an important part of dealing effectively with complex systems. As a part of this ability, Ada provides encapsulation mechanisms which make the development of software libraries easy and natural. But in addition to encouraging the development of new libraries, for a new language to be attractive to old programmers, it must also have a way to immediately make libraries available which are at least as good as the mature ones currently available in other languages. For example, the scientific work currently done in FORTRAN would be impossible without the valuable, mature math libraries available with almost any FORTRAN implementation. Ada has an answer for that with its ability to define interfaces to other languages. With this ability, existing libraries from virtually any language can be made available to Ada programs, thus providing for the needed transition period from older languages to Ada. [5, 11]

The encapsulation of features in libraries is an important concept for system maintenance, as it isolates system features with well-defined interfaces from other parts of the system. Also, as more mature Ada libraries become available to replace older libraries in other languages, the insertion of the new libraries could become a part of maintenance activities. Such replacements should be relatively straightforward for the maintainer since a library has well-defined interfaces.

In addition to being able to work with high levels of abstraction, it is also necessary at times for embedded software to be able to access machine details. Features built into Ada provide this capability. However, because of Ada's emphasis on readability, even machine level programming details can be understandable to anyone reading an Ada program. This is an especially important feature to provide for maintainability of the software. [5, 11]

In this time of increasing performance demands on software, it is also important for many embedded systems to be able to do concurrent processing. This is one area in which Ada has gone far beyond most other languages. Ada has built in structures for dealing with concurrent processes, along with a sophisticated mechanism for communication among these processes. Although concurrency is by no means an easy concept for either development or maintenance, handling it at a relatively high level of abstraction is much easier for any human to deal with than the more common method of using the lowest level of machine instructions. [21, 5, 11]

Finally, it is well-known that computer systems do sometimes fail. Sometimes the hardware is at fault and sometimes the software, but more importantly, sometimes the system supports critical functions which just must not fail. The Ada designers recognized that fault-tolerance is an important issue, and so the language provides a specific mechanism, exceptions, for dealing with "unusual" or exceptional conditions which can cause system failure. This is an important feature for corrective maintenance because it makes problems easier to track. It is just as important, however, that exceptional conditions be considered and "covered" any time modifications are made during maintenance. [5, 11]

More Than Just a Language

By the time the development of the Ada language was well underway in the late 1970s, it was already recognized that the magnitude of the DoD's software problems was enormous. A programming language alone would not be sufficient to solve all of these problems. The entire life cycle process had to be improved. Thus began the effort to define a programming support environment to go along with the language and provide it with quality support tools to support the entire development life cycle. [5] This initial effort ended with a requirements document for an Ada Programming Support Environment (APSE), and the document became known as "STONEMAN" [10].

As stated in STONEMAN, "The purpose of an APSE is to support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications." [10] The basic concept of the APSE is to provide an integrated set of portable tools which can be improved and extended as time goes on. These tools would not only be able to support the technical aspects of software development, but the

These tools would not only be able to support the technical aspects of software development, but the management aspects as well. And because embedded systems are typically so small that they may not be able to support the entire APSE during development, an important part of the APSE concept is to have both a host and a target system. The host is the system on which the software is developed, and the target is the system on which it will eventually run. Of course, any given system may be both. But the important thing is to provide a good development environment for the software which will eventually run on the embedded target. [5]

Portability is a key issue in the development of an APSE. If tools can be made portable, then once developed they can be used in any APSE, even if implemented on different hardware. Of course, such a powerful concept is not easy to achieve, but the APSE framework laid out in STONEMAN provides a basis for achieving it. The idea is to encapsulate the system hardware and system-dependent software by a common set of interfaces. This provides a way for any APSE tool to access the facilities provided by the system in a standard way. Thus, the tool does not have to be dependent on a particular system in any way, just on the set of interfaces. What is necessary is for a standard set of interfaces to be defined and for the actions defined by these interfaces to be essentially the same on every implementation. [10, 5]

Such a set of interfaces has been defined by the DoD. The first official version of the Common APSE Interface Set (CAIS) was published in October of 1986 [12] with the intention that it be used in the beginning for research and development purposes. It is expected that experience will help this standard to develop into something that is acceptable to both implementors and users. Meanwhile, the DoD already has a validation capability for the CAIS under development. [14] Hence, once the CAIS standard is mandated by the DoD, a mechanism will be available to determine conformance to this standard in much the same way conformance to the Ada language standard is determined.

In addition to promoting portability, the APSE concept also calls for the entire set of tools to be integrated. This concept permits tools to use other tools, keeping continuity in the software development activities, rather than requiring the programmer (designer, project manager, etc.) to explicitly stop one activity and start another every time a different action is required. [10] It also permits an underlying structure to keep track of the activities of many working on the same, or even multiple, projects without disturbing those activities. This allows for both information gathering and project control activities. It can also provide enforcement of certain procedures if this is deemed desirable. [18]

Because the APSE has been defined to be extended, it is not yet determined exactly what tools will be developed for it beyond those tools standard in today's software development organizations (compiler, editor, debugger, linker/loader, etc). As Booch indicates, this provides a great potential for innovation. He divides the potential tools into two classes: generic and methodology-specific tools. Generic tools will support programming tasks in general, without regard to specific disciplines. Methodology-specific tools, on the other hand, will support a particular programming or management discipline. [5]

Since maintenance activities depend upon the management and technical activities involved in the original development, the APSE will directly support maintenance as well as software development. The support for orderly development will provide more pertinent system information to the maintainer, and, perhaps even more importantly, that information can be consistent and complete. Tool portability will perhaps even put the maintenance organization in better shape than the typical development organization, for the maintenance APSE can consist of a conglomeration of all tools which have been used on numerous developments. These APSE tools can provide the maintainer with tremendous productivity benefits.

Criticisms of Ada

A discussion of Ada with respect to software maintenance would be incomplete if it only included Ada's strengths and did not address some of its criticisms. Language weaknesses may have some bearing on the maintenance of systems implemented in Ada. The weaknesses of any language must be considered when determining what is and is not possible in an implementation of that language.

The two most notable criticisms of the Ada language at present are its size and its concurrency model. Whether the size of the Ada language definition is a problem or not is a matter of current debate. Rather than argue one way or the other, let's just look at a potential maintenance problem. It is obvious that a larger language requires more time for a software professional of any type to learn than a smaller one. However, to be fair to Ada, it should also be noted that Ada requires more time to learn for another reason as well. Ada was developed to be more than "just another language". In order to use Ada as intended, a software professional must also understand and use the principles of software engineering. Hence, to learn the "Ada technology", one must not only learn a large language, but also how the principles of software engineering can be applied with that language.

Additionally, though Ada may take longer to learn, it could turn out to be one of a very few languages, or possibly even the only language, a maintainer needs to know. This could mean a smaller amount of learning time spent in the long run, since the learning process would not have to be repeated numerous times for other languages.

As for the concurrency model, the concern is that it may not be adequate to permit the necessary performance required of some real-time systems. This is a legitimate concern, and it is already being addressed at the Software Engineering Institute [15]. As noted above, concurrency is no small problem for any software system to deal with. However, the use of a higher level of abstraction, where possible, when dealing with concurrency is still a plus for the maintainer. One of the most significant parts of the maintenance task is to understand the system and how it works. This is facilitated by the ability to see the system at the high level of abstraction.

Many other common criticisms of Ada are actually criticisms of specific implementations rather than language criticisms. Since the state of most Ada systems software is less mature than that of other languages which have been around longer, criticisms such as language inefficiency have been common. However, the inefficiencies have been in the implementations, and they are not inherent in the language definition. In fact, the efficiency of most implementations is improving at a rapid rate.

Probably the most common implementation criticism still blamed on the Ada language is that of its input/output (I/O) capabilities. The I/O was purposely defined by the language in terms of very basic building blocks, effectively permitting limitless possibilities for the development of very sophisticated I/O libraries. Unfortunately, vendors have yet to take much advantage of the building blocks available. This could mean additional requirements for system development, or it could also mean enhancements required as a part of system maintenance.

This brings us back to system development activities. But development activities include management as well as technical activities, and both types have significant impact on the maintenance of a system. The following sections will look at both types of activities, how they are supported by Ada, and how that support affects software maintenance.

II. Management of Development Activities

Management of a software development project includes numerous activities, many of which have significant impact on future maintenance of the software developed. Specific management concerns include project control, quality control, and configuration management. Ada can provide support in each of these areas, and this support can have a positive effect on software maintenance. The basis for this is the automated support provided by the Ada environment, the APSE.

Unlike computer languages which are simply implementation tools, Ada can provide numerous management benefits which support the technical work. The APSE can be a powerful management tool, permitting most management functions to be automated. This will not only cut the time required for performing certain functions, but in many cases it will also provide for better results [22].

To begin with, an integrated environment such as we find in the APSE concept (it is not a part of the state of the practice in most APSEs yet), has ultimate control over all software development activities taking place. Although a manager must be very careful about the psychological effects of how this control is perceived by the software developers [18], this permits the automatic collection and structuring of data for many purposes.

The concept of an integrated environment is not unique to the APSE. In Europe, there has been much interest recently in the integrated project support environment (IPSE), a concept which is not language specific. In fact, an APSE is just an IPSE for Ada [19]. It is unfortunate that the "P" in APSE stands for "programming" rather than "project", for an Ada project support environment is really a better description of an APSE. Nevertheless, the Europeans are applying their IPSE concepts to developing APSEs, just as we are in the U.S. Ada has become a standard in a number of European countries, in addition to being a standard for the North Atlantic Treaty Organization (NATO). [5, 9]

Project Control

An important part of project control is knowing the current status of the project. This is often very difficult information to obtain when the subject is a software project, but an APSE can change that completely. If the project uses automated tools for all parts of the life-cycle, and these tools are a part of an integrated APSE, then the APSE can collect data on every aspect of the project automatically. Although knowing what type of data to collect, how to collect it, and what to do with it afterward is an important area needing more study, it is clear that the right type of data collection could provide the project manager with far better data than can be obtained just by asking for subjective information from the people working on the project. And the automatic collection of data has several important side benefits. It will improve productivity because people will not have to spend time providing status information to management, and it will also improve the historical database of information which will be important for both maintenance of the current project and for planning future projects. [18, 22]

The potential importance of the information which can be automatically and systematically collected by an APSE cannot be understated when considering the maintenance function. Current maintenance difficulties are mostly a result of poor software engineering practices, much of which are manifested in poor or missing documentation. One of the capabilities of the APSE, given the right integrated tools, is the automatic and systematic collection and modification of system documentation.

Quality Control

The data collected by the APSE can also be of great benefit to the software quality assurance (SQA) function of a project. Any SQA program requires the development of standards. One of the biggest difficulties with SQA is determining if these standards are being followed. Once established, appropriate automated functions can check against these standards very easily, providing a major step toward SQA effectiveness. [8, 7]

An effective SQA function is another basic ingredient required for maintaining good software engineering practices in a software project. The standardization of practices is what makes systematic system documentation possible.

Configuration Management (CM)

Another area which can benefit greatly from the APSE is configuration management (CM). This is an important area for assisting with project control, yet it is often not very effective in practice. One of the reasons for this is probably that doing CM without automation requires a major effort, and it is also difficult to keep the CM current with the ongoing progress of the software development. An automated process integrated into an APSE can change this, making CM an effective part of project management. [4, 22]

With such consistent automated control over the system configuration, maintenance is once again affected. Without good CM, module changes can impact unexpected areas and even different version releases. However, automated CM can prevent this. Together with effective software engineering practices, as discussed below, CM automated via an APSE can turn maintenance into a systematic discipline of software modification.

III. Technical Development Activities

The management activities of software development, described above, are complemented by the technical activities, described in this section. Any software development project requires both, and both types of activities have important effects on maintenance. Technical activities are supported both by features of the Ada language and features of the APSE.

The APSE Interactive Monitor (AIM) project developed for the Naval Ocean Systems Center is an example of a project benefiting from Ada's software engineering features. The design of the AIM project took longer than would be expected, based on representative life-cycle models. However, testing time was much lower than expected. AIM was successfully ported from the system on which it was developed to another entirely different hardware/operating system pair with minimum effort. And it was delivered both under budget and ahead of schedule. Its success was largely attributed to the use of Ada. [3]

The various software engineering features of Ada which contribute to such successes include reliability, reusability, portability (also called transportability), and methodology and life-cycle support. Each of these is discussed below, along with its effect on maintenance.

Reliability

For software to be considered reliable, its user expects it to work "correctly" virtually all the time. But even "correct" has different meanings in different situations. Sometimes it just means that the software will never "blow up" or cause the user to lose valuable time or information. Other times it means that the software will meet certain stringent conditions and/or it will never permit an unsafe or life-threatening situation to occur. [5]

Experiences with systems such as AIM give credence to Ada's claims to support reliability as well as maintainability. The fact that AIM system testing required so little time, when compared with other similar projects done in other languages, was attributable to the relatively small number of errors found during testing [3]. This is a good indication that those software engineering features are doing their job in making errors easier to spot and correct early in development. As a result, better software reliability is believable.

The sheer readability of Ada code, combined with the fact that Ada supports expression at the "proper" level of abstraction throughout a program (because it supports all levels), also supports reliability and maintainability. For if a program can be understood, then it can be safely modified. Of course, ease of modification is also greatly aided by modularization. Just as modifying the workings of a machine is easier when the machine has distinctly identifiable parts which interact in well-known ways, modifying software is also easier when the software has distinctly identifiable modules which interact in well-known ways. This type of structuring is greatly facilitated by the features of Ada. [5]

This is not to say that poor code cannot be written in Ada. A good tool in the wrong hands or used for the wrong purpose can certainly result in a poor product, and the same is true for Ada. But if software engineers are truly making progress in their relatively young discipline, then the features built into Ada should make a significant difference in reliability and maintainability. The key is for Ada to be used by those who understand the principles of software engineering.

Reusability

Another important software engineering feature of Ada is the potential for reusability [5]. Because of the ease with which libraries may be created in Ada and the fact that the Ada structure most often used to create them is called a package, they are often referred to as packages rather than libraries. But the terminology doesn't matter. The important thing is that any unit which can be put into an Ada library has the potential for reusability, and putting units into a library is an automatic part of Ada development. Hence, the potential is tremendous.

Once again, just because the potential is there doesn't mean it will be used. And many issues must be considered in order to write software which can be reused easily. However, much has been written on how to create reusable software, and many are specifically detailing how this can be done in Ada. Gargaro and Pappas, for example, demonstrate the differences among weak reusability, strong reusability, and effective reusability. They emphasize that many factors must be considered, but they also illustrate that Ada is a good tool to use for writing reusable software [16]. Booch discusses three "levels" of reusability -- packages, tools, and subsystems. Subsystems are typically 20,000 - 30,000 lines of documented Ada software developed in an object-oriented manner [6].

The importance of reusability to maintainability depends on the amount and type of software which is reused. In general, the reused software would usually be expected to be more mature software, which would theoretically require less maintenance. However, reusability has not become enough of a reality yet to make such clear-cut statements. Certain attempts at reuse could be incorrect uses of the software, and this could result in additional maintenance problems.

If reuse is accomplished via Ada libraries of package structures or Ada subsystems, then much of the potential maintenance difficulty of software reuse can be avoided. Proper packaging provides for encapsulation of abstractions and avoids undesirable module coupling. This in turn provides for much easier maintenance, free of undesirable side-effects on other parts of a software system.

Portability, the Reusability of APSE Parts

Other software engineering benefits of Ada can be realized with the use of an APSE. The APSE structure is built on the concept of portability, a high level form of reusability [10]. Portability implies that entire programs can be reused on different systems. For an APSE, this means that entire tools or even tool sets can be reused. This is a powerful concept. It means that instead of having at most a few tools available for a new project, entire tool sets can be available for every new project. Hence, every project can have the benefit of a set of mature, established tools. The implications this has for productivity are tremendous, but it's just a beginning.

Imagine a large organization developing a set of APSE tools which can be ported to any of numerous types of computer systems. Once developed, they can be used on virtually any project. Not only is the potential cost savings for the organization tremendous, but it frees the ingenuity of software engineers and designers to consider higher level, more powerful tools. These could be built on the base of the established tools. With such powerful tools, the productivity implications begin to multiply. And the only limit is in the imagination of those developing the tools.

As for maintainability, a good set of tools is as important to the maintainer as it is to the developer. Furthermore, the ability to use the same set of tools on numerous systems provides a continuity among systems. This can be very important to the maintainer who is responsible for several systems at the same time. It not only provides for a more effective tool set, but it also saves considerable time usually spent in adjusting to a new tool set every time a system switch is made. And even if the tool set is not exactly the same on each system, just using the same language and software engineering concepts throughout is an important continuity. The importance of life-cycle continuity is discussed below.

Methodologies and Life-Cycle Support

Further software engineering benefits of the APSE concept are the ability to build in complete life-cycle support for any software development methodology and the ability to define the life-cycle in many ways. Many different methodologies are used today, but few, if any, have automated support throughout the life-cycle. Yet the use of a methodology is an important software engineering feature [5].

Although no one methodology is universally used today, some particular application areas and methods are beginning to get a lot of attention. For example, sequential methods don't typically work well with concurrent applications, so many new methods have been developed recently for working with concurrency [21]. Also, many see object-oriented design as the way of the future of software engineering, a fundamental requirement for fostering software reusability and being able to handle large, complex systems [20]. Duff believes the object-oriented model will be a unifying force. He states, "The object-oriented model will assimilate many of the specialty areas that are now discrete, including data bases, financial modeling, logic programming and other areas of AI, graphics, text formatting, and computer-aided design." [13] The object-oriented design method has also won considerable support in Ada circles [5].

The traditional life cycle is defined by a "waterfall" model based on strictly top-down hierarchical decomposition. It is recognized that this is a good method for developing software if the solution is known, but many believe it to be a poor way to develop new types of software for new applications. A number of new paradigms are now being used for software development, including rapid prototyping and operational specification. [23, 2]

The advantage to the APSE approach to tools is that the tools can provide support for any desired life-cycle model in combination with a desired methodology. Or, an APSE can even contain several sets of tools supporting multiple methodologies and life-cycle models, and a project can choose the best combination for a particular application.

In any case, the continuity of complete life-cycle support is important for system maintenance. Many modifications require changes to the original system requirements, and then the effects of the change ripple through the entire life-cycle. Such changes would be greatly facilitated by automated support for following the ripples all the way from documentation changes to actual source code modifications.

IV. Conclusion

Now that we have examined the match between Ada and the needs of software maintenance, it has become clear that Ada has much to offer both in language features and life-cycle support. The APSE concept provides just the right framework for supporting all of the basic management activities. It provides the capabilities for automated monitoring, configuring, and data collection to support the many activities involved with project control. It also answers the needs for enhanced productivity and system reliability. All of these features not only make a system easier to develop, but also to maintain.

As far as technical issues, perhaps Jean Ichbiah, Ada's chief designer, best summarizes the future expectations of the language in this area:

> "Ada has all the right facilities. The package concept embodies the software engineering principles needed for organizing and building large systems. The tasking features allow you to express an application's inherent parallelism directly in your program. The program library concept maps well to distributed host environments, allowing networked team members to share common libraries. Regarding the software development process itself, programs will be constructed much more by composition out of existing components than by developing them from scratch. On-line data bases of Ada packages will be accessible for this purpose." [17]

All of these software engineering features supporting technical development activities provide a solid basis for software maintenance. Ada is not the perfect language, but it is the best currently available. And the APSE makes it more than just a language, but rather a technology capable of complete software engineering life-cycle support. All in all, Ada technology far surpasses the capabilities of any isolated language in providing for both effective system development and its subsequent maintenance.

References

[1] Ada Information Clearinghouse, List of Validated Ada Compilers, *Ada Information Clearinghouse Newsletter*, Vol VI No 1, March 1988.

[2] William W. Agresti, "What Are the New Paradigms?", in *New Paradigms for Software Development*, edited by William W. Agresti, IEEE Computer Society Press, 1986.

[3] Jerry Baskette, "Life Cycle Analysis of an Ada Project", *IEEE Software*, Vol 4 No 1, January 1987.

[4] Rudy Bazelmans, "Evolution of Configuration Management", *Software Engineering Notes*, Vol 10 No 5, October 1985.

[5] Grady Booch, *Software Engineering with Ada*, 2nd edition, Benjamin/Cummings, 1987.

[6] Grady Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[7] Martha Branstad and Patricia B. Powell, "Software Engineering Project Standards", *IEEE Transactions on Software Engineering*, Vol SE-10 No 1, January 1984.

[8] Fletcher J. Buckley and Robert Poston, "Software Quality Assurance", *IEEE Transactions on Software Engineering*, Vol SE-10 No 1, January 1984.

[9] Virginia L. Castor, "Dramatic Progress", *Defense Science & Electronics*, Vol 5 No 3, March 1986.

[10] Department of Defense, *Requirements for Ada Programming Support Environments*, "STONEMAN", February 1980.

[11] Department of Defense, *Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

[12] Department of Defense, *Common Ada Programming Support Environment (APSE) Interface Set (CAIS)*, DOD-STD-1838, 9 October 1986.

[13] Charles Duff, "Programming in the 1990s", *Computer Language*, Vol 4 No 12, December 1987.

[14] APSE Evaluation & Validation (E&V) Team, MILNET electronic mail communications, 1987.

[15] Robert Firth, "A Pragmatic Approach to Ada Insertion", Proceedings of the International Workshop on Real-Time Ada Issues, *Ada Letters*, Vol VII No 6 (Special edition), Fall 1987.

[16] Anthony Gargaro, "Reusability Issues and Ada", *IEEE Software*, Vol 4 No 4, July 1987.

[17] Jean Ichbiah, "Programming in the 1990s", *Computer Language*, Vol 4 No 12, December 1987.

[18] B. A. Kitchenham and J. A. McDermid, "Software Metrics and Integrated Project Support Environments", *Software Engineering Journal*, January 1986.

[19] John McDermid, Introduction by the editor, *Integrated Project Support Environments*, Peter Peregrinus Ltd., 1985.

[20] Bertrand Meyer, "Reusability: The Case for Object-Oriented Design", *IEEE Software*, Vol 4 No 2, March 1987.

[21] Kjell W. Nielsen and Ken Shumate, "Designing Large Real-Time Systems with Ada", *Communications of the ACM*, Vol 30 No 8, August 1987.

[22] Paul Rook, "Controlling Software Projects", *Software Engineering Journal*, January 1986.

[23] Yu Wang, "A Distributed Specification Model and Its Prototyping", *Proceedings of COMPSAC'86*, October 1986.