Technical Report

CMU/SEI-92-TR-029
ESC-TR-92-029

Ada Adoption Handbook: A Program Manager’s Guide

Version 2.0

William E. Hefley
John T. Foreman
Charles B. Engle, Jr.
John B. Goodenough

October 1992

Technical Report
CMU/SEI-92-TR-029
ESC-TR-92-029
October 1992

Ada Adoption Handbook: A Program Manager's Guide

Version 2.0

William E. Hefley
SEI Services

John T. Foreman
Defense Advanced Research Projects Agency

Charles B. Engle, Jr.
Florida Institute of Technology

John B. Goodenough
Real-Time Systems Program

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Preface to the Second Edition

This new edition of the *Ada Adoption Handbook: A Program Manager's Guide* extensively updates the original 1987 SEI report [109]. Since that time, over 1,500 copies of the first edition have been distributed by the SEI, and almost 500 copies by NTIS and DTIC, making this one of the most-frequently requested documents in DTIC holdings. It is impossible to estimate the number of copies of the first edition that have been reproduced and distributed by readers.

In the intervening years, the technology supporting Ada has improved dramatically, Ada has come into wider use, and the mandate to use Ada for defense systems has been strengthened. This second edition is motivated by the need to update the earlier version to address current issues regarding the use of Ada. In 1987, the issues revolved around the technical adequacy of Ada, its tools, and its initial adoption and use; in 1992, the compilers and tools have greatly improved and a much larger usage base has provided experience and results to better understand the issues of adopting Ada and managing Ada projects. Today, the issues emphasize the management aspects of using Ada in the development of software-intensive systems.

The intent of this second edition is to help program managers become better consumers of Ada products and systems by understanding:

- the management, personnel, and technical implications of Ada
- Ada's role in the software development process

Understanding these topics will aid managers in moving their organizations toward widespread use of Ada in building and maintaining software-intensive systems.

Acknowledgements

As part of the development of this document, we received assistance, reviews, and comments from a wide spectrum of people, including:

- Ada technologists
- Department of Defense (DoD) policy makers
- system developers and support organizations
- DoD and industry program managers

We thank them all. We are also grateful to the following individuals who gave so generously of their time:

Neal Altman, SEI
Christine M. Anderson, Phillips Laboratory, USAF
Judy Bamberger, Loral Western Development Labs
Rick Barbour, SEI
Capt. Jim Cardow, USAF,
Air Force Institute of Technology
Susan Carlson,
Ada Information Clearinghouse
Currie Colket, Ada Joint Program Office
Pat Donohoe, SEI
Les Dupaix, Software Technology Support
Center, USAF
Michael Feldman, George Washington University
Robert Fritz, Orincon Corporation
Dale Gaumer, Magnavox Electronic Systems
Company
Col. Rick Gross, USAF
Audrey Hook, Institute for Defense Analyses
Steve Jacobs, TRW
Kent Johnson, Software Productivity Consortium
Reed Little, SEI
Steven Litvintchouk, MITRE
Robert Mathis
Nancy Mead, SEI
Geoff Mendal,
Systems Engineering Research Corporation
Jim Moore, IBM
Cdr. Melinda Moran, USN, Naval Computers and
Telecommunications Command
Patricia Oberndorf, Naval Air Warfare Center
John Pates, SEI
Tom Ralya, SEI
Gordon Smith, Southern Methodist University
Doug Waugh, SEI

The authors express special appreciation to several of our colleagues for their exacting reviews and pragmatic insights: Nelson Weiderman, Dan Roy, and Daniel Burton, all of the Software Engineering Institute.

This second edition of this handbook has also benefited from a tutorial seminar *Adopting Ada: Management Perspective* that has been presented at a number of conferences and meetings, including SIGAda and TRI-Ada, and as an elective within the Program Manager's Course at the Defense Systems Management College. The tutorial is derived from the two volumes of the *Ada Adoption Handbook* series, specifically: *A Program Manager's Guide* [109] and *Compiler Evaluation and Selection* [226]. John Goodenough, John Foreman, Dan Burton, Judy Bamberger, and Bill Hefley contributed to the various versions of this seminar. Our heartfelt thanks go to all the attendees and students who have indirectly contributed by asking hard questions and bringing their concerns with them.

Additionally, Sandra Bond (Software Engineering Institute Information Management group) has been invaluable throughout the preparation of two editions of this handbook for trying to "make every word count."

Ada Adoption Handbook: A Program Manager’s Guide

Abstract. The *Ada Adoption Handbook* provides program managers with information about how best to tap Ada’s strengths and manage the transition to fully using this software technology. Although the issues are complex, they are not all unique to Ada. Indeed, many of the issues addressed in this handbook must be addressed when developing any software-intensive system in any programming language. The handbook addresses the advantages and risks in adopting Ada. Significant emphasis has been placed on providing information and suggesting methods that will help program and project managers succeed in using Ada across a broad range of application domains.

The handbook focuses on the following topics: Ada’s goals and benefits; program management issues; implications for education and training; software tools with emphasis on compiler validation and quality issues; the state of Ada technology as it relates to system design and implementation; and the pending update of the Ada language standard (Ada 9X).

1. Introduction

1.1. Purpose and Scope

Ada is a programming language for developing software, including information systems, mission-critical, and embedded applications. This handbook presents program managers with information to make effective use of Ada.

This handbook[1] provides program managers with information about how best to tap Ada’s strengths and incorporate this software technology into their programs. Although the issues are sometimes complex, adopting Ada can be managed. Similar issues must be addressed when using any programming language for building sophisticated software-intensive systems. Although Ada’s origins were strongly influenced by the needs of the defense mission-critical computing community, the language design team placed emphasis on supporting modern software engineering practices; the result is a language with wide applicability in developing well-engineered, quality software.

This handbook addresses the advantages and risks inherent in Ada adoption. Significant emphasis has been placed on:

- providing information and suggesting methods that will help program and project managers succeed in using Ada, and
- presenting that information in an objective manner.

This handbook should not be construed as supplanting DoD policy.

This handbook has been written for use across many application domains and covers a large number of fundamental Ada issues. However, it does *not* address issues that are unique to individual application domains (for example, ground-based command and control systems or embedded avionics) or specific programs.

[1]This handbook is one of two volumes of the *Ada Adoption Handbook* series produced by the SEI, which also includes the *Ada Adoption Handbook: Compiler Evaluation and Selection* [226]. The *Compiler Evaluation and Selection* volume provides a substantive overview of the issues confronted by software managers and lead technical personnel tasked with evaluating and selecting Ada compilers.

1.2. Handbook Overview

This section describes the organization of this handbook.

This handbook is organized as stand-alone chapters that address program managers' questions about Ada and provide plans for adopting and inserting Ada into an organization. The chapters are:

1. **Introduction**: This chapter. Includes the purpose of this handbook and provides guidance for using the handbook.
2. **Program Manager Considerations**: Commonly asked questions, succinct answers, and, where needed, pointers to more detailed information. Topics include general information, technical, management, cost, and program control issues. This chapter can be used (among other purposes) as an executive summary for the rest of the handbook and to review particular points.
3. **The Need for Ada**: An overview of Ada's goals and benefits, and policies for Ada's use.
4. **Management**: An overall view of the impact of Ada on the management process and management factors relating to the transition to Ada use.
5. **Learning Ada**: Training implications and education issues.
6. **Software Production Technology**: An overall view of software tools, with particular emphasis on compiler validation and compiler quality issues.
7. **Ada Applicability**: An overall view of the current state of Ada technology, and discussion of system engineering issues.
8. **System Design and Implementation Issues**:
    - **Designing for Portability and Reuse**: Discusses how Ada facilitates developing portable and reusable software.
    - **Using Ada with Other Standards and Protocols**: Using Ada interfaces to other systems and standards for successful integration of Ada software in overall systems environments.
    - **Using Special-Purpose Languages**: The application of special-purpose languages such as ATLAS, simulation programming languages, LISP (or other artificial intelligence languages), and fourth-generation languages.
    - **Mixing Ada with Other Languages**: Possible approaches for dealing with existing operational systems, including language translation and hybrid (mixed language) systems.
9. **From Ada 83 to Ada 9X**: A discussion of the Ada 9X Project, the evolving Ada 9X language standard, and the expected impact on present and future programs of moving from the present language standard (Ada 83[2]) to the revised standard (Ada 9X).

The following appendices are also included:

A. **Ada Information**: Recommended readings and resources for individuals looking for fundamental Ada information.

B. **Ada Working Groups and Resources**: Information about organizations involved in Ada issues, standardization, and policy.

[2]Although the International Standards Organization (ISO) standard was approved in 1987, the existing Ada language is often called Ada 83, at least in the United States, as the American National Standards Institute (ANSI) standard [13] was approved in 1983.

1.3. Tips for Readers

This section highlights techniques for quick and efficient use of this handbook.

In addition to the question-and-answer approach used in Chapter 2, several other techniques have been used to help the reader make maximum use of this handbook:

- **Summaries**: A summary begins each major section. Each summary is centered and italicized for easy identification.
- **Bold and Bullets**: Major points are emphasized by using bold headings within bulleted lists. The major points are then followed by detailed discussion.
- **Action Plans**: Actions to facilitate the adoption of Ada are presented at the end of some major sections.
- **Reading Lists**: Detailed additional material on specific topics are identified at the end of some chapters.
- **Software Development Tutorial**: The terms and concepts of software development discussed in this handbook are defined in Section 6.1. Readers unfamiliar with these terms and concepts are strongly encouraged to read that section first.

The following topics are not covered in this handbook:

- Details of the Ada language or software engineering concepts.
- Details of particular methodologies such as structured analysis, object-oriented design (OOD) or any specific OOD techniques such as hierarchical object-oriented design (HOOD); other design or analysis methods such as process abstraction method (PAMELA), Ada-based Design Approach for Real-Time Systems (ADARTS), or rate monotonic analysis (RMA).

1.4. Changes in Version 2

This version of the handbook has been extensively revised to reflect the current state of Ada usage. This section summarizes those revisions.

Although the Ada language definition has remained constant since its standardization in 1983, much has changed with respect to the Ada milieu since the previous version of this handbook — the number and quality of compilers have increased, and many additional efforts have used Ada successfully in the intervening years. This edition has been revised and expanded to incorporate some of the lessons learned since 1987. Major changes include:

- **Chapters 2 Program Manager Considerations** and **Chapters 3 The Need for Ada** have been updated.
- **Chapter 4 Management** has been added.
- **Chapter 5 Learning Ada: Training Implications** has been extensively revised.
- **Chapter 7 Ada Applicability** has been updated.
- **Chapter 8 System Design and Implementation Issues** has added material about Ada bindings to other standards and protocols, as well as material on portability.
- **Chapter 9 From Ada 83 to Ada 9X** has been added to introduce the Ada 9X efforts and the enhanced capabilities that are defined in the evolving Ada 9X standard.

- **For More Information** sections containing reading lists have been added to several chapters.
- **Appendix A Ada Information** has been added to provide the reader with additional sources of information about Ada.
- **Appendix B Ada Working Groups and Resources** has also been updated.
- An acronym list has been added to aid the reader.
- The reference list has been extensively updated.
- Appendices on Ada textbooks, Ada compilers, and Ada programs have been deleted because more current information can be obtained from the Ada Information Clearinghouse or the *Language Control Facility Ada-JOVIAL Newsletter*. See Appendix B for contact information.

2. Program Manager Considerations

Ada offers considerable advantages over other languages used for software development. Because Ada technology and its use are still evolving, program managers may have questions about this technology. Questions fall into several categories: general, costs, technical issues, program management, and getting help.

This handbook provides program managers with information to make well-informed decisions about using Ada. Some frequently asked questions and answers are presented on the following pages. Where needed, pointers to supplemental information and appropriate actions are provided.

2.1. General

The following questions are covered in this section:

- What is Ada and why was it developed?
- For what application domains should Ada be used?
- What is DoD policy on Ada use?
- What are the advantages of using Ada?
- What are some of the inhibitors to adopting Ada?
- Who manages Ada?
- What is the Ada 9X Project? What will its impact be?

Question: What is Ada and why was it developed?

Answer: Software development and post-deployment support are increasingly recognized as serious problems for software-intensive system development. Ada, the computer programming language (ANSI/MIL-STD-1815A [13]) developed under the sponsorship of the Department of Defense (DoD), addresses these problems by:

- reducing the need for multiple programming languages, so effort can be focused on supporting fewer languages and making one language work well, and
- enabling the use of modern software development methods, thereby reducing the costs and risks of software development and facilitating improved maintenance during post-deployment support.

Ada is a United States, North Atlantic Treaty Organization (NATO), international, and DoD standard programming language. The Ada language standard does not itself define software development methods and processes for using that programming language.

See: Sections 3.1, 3.2.

Question: For what application domains should Ada be used?

Answer: Ada is a programming language designed to meet the needs of developing and maintaining large, complex software systems. Although Ada was originally designed to support real-time and embedded systems, usage has shown that Ada is applicable to many other application domains, such as large-scale information systems, scientific computation, and systems programming. Ada has been successfully used in applications that vary widely in size and application domains in both the public and private sectors.[3] In domains such as scientific computing or management information systems (MIS) applications, a number of existing utilities (numeric libraries, report generation tools, etc.) may have to be re-created or interfaced with to achieve with Ada the same degree of usefulness that they had with other languages.

The development of the Ada language focused programming development methods and tools on a single language that supports modern software engineering techniques. These techniques are applicable regardless of the application domain.

See: Sections 3.3, 4.1.1.2, 7.1.

Question: What is DoD policy on Ada use?

Answer: By legislation and policy, Ada has been defined as the required language for developing Department of Defense software development, where cost effective. Ada's role as the single, common, high-order programming language is a major step forward in addressing DoD software development problems. Current DoD policy regarding the use of Ada is described in DoD Directive 3405.1 [77] and DoD Instruction 5000.2 [79]. This latter instruction [79], which implements the legislation mandating Ada's use within the DoD, requires that Ada be the only programming language used in new defense systems and major software upgrades (the redesign or addition of more than one-third of the software) of existing systems.

This policy places the responsibility of applying the Ada mandate on the individual program manager — to comply, managers will have to be aware of issues such as appropriate hardware and software selection (to ensure that validated Ada implementations are selected), implications of life-cycle costs, and the size or magnitude of planned systems or upgrades.

Each DoD component has designated a senior official as the Software Executive Official, who will serve as a focal point for Ada usage within its organization.

See: Sections 3.2, 6, 7, Appendix B.4.

Question: What are the advantages of using Ada?

Answer: Ada's expected overall advantage can be summarized as increased quality per dollar spent. Specifically, benefits have been reported in productivity, software portability, portability of programmers among projects using a single language, and software maintainability and reliability [193, 29, 107, 191]. An area where Ada's impact has been positive is in increased productivity. While productivity may decline by 10 to 20 percent during the first two to three Ada projects, productivity after complete transition to Ada has been shown to increase on average about 20 percent [193]. Ada projects seem to concentrate greater effort in the early stages of the software development process and have achieved sizable reductions in error rates — signs of improved software engineering practices that have been facilitated through the use of Ada. Additionally, Ada provides a common basis for using tools and methodologies along with new capabilities for managing system complexity. Cost savings accrue in terms of the numbers of unique configurations needed to maintain software written in Ada.

See: Sections 3.1, 3.3, 4.1.1.2.

[3]Information identifying many efforts that have used Ada, both defense-related and commercial, is available from the Ada Information Clearinghouse Ada Usage Database. (See Appendix B.3 for contact information.)

Question: What are some of the inhibitors to adopting Ada?

Answer: The following inhibitors have been encountered in adopting Ada:

Compiler availability: Most major processors in use today, ranging from specialized digital signal processors (DSP) to microprocessors to mainframe computers and super computers, have Ada compilers. There were 501 *total* validated Ada compilers on the official AJPO list (as of October 1992). This number has grown from 78 validated compilers in May 1987, and only 14 in early 1986. See Appendices A.2 and B.3 for online and printed sources of the current listing. If no compiler is available for the selected hardware, see Section 7.2.1 for an action plan and several alternative solutions.

Ada and embedded systems: The Ada language design team emphasized supporting modern software engineering practices; the result is a language with wide applicability in developing well-engineered, quality software. In fact, Ada has been used successfully for MIS and Corporate Information Management (CIM) applications [87]. There are no technical reasons why Ada cannot be used successfully, and cost-effectively, for such applications [64, 87].

DoD policy and Ada: Current DoD policy requires that Ada be used for new defense systems and for major software upgrades of existing systems, where cost effective. See Section 3.2 for a brief description of the waiver process for efforts that cannot comply with the policy.

New technology: A new technology always introduces risks, but now that Ada has matured, the risks from adopting Ada have been significantly reduced. Recent studies have shown that, in organizations that have completed several Ada projects, Ada can be at least as cost-effective, if not more so, as other languages that have traditionally been used for developing large, software-intensive systems [39, 193, 81].

Lack of knowledge: A lack of knowledge of software engineering and Ada can delay the transition to Ada. Software engineering has not yet attained the recognition or acceptance of other academic disciplines. Education in software engineering is not as available, comprehensive, or complete as in established engineering disciplines. An effective training program is a key part of developing an organization’s software engineering capability [147]. Ada training, supported by appropriate software engineering training, can assist an organization in improving that capability.

DoD procurement process: The current procurement process may not be conducive to Ada adoption and long-term software engineering improvement. A recent survey of Ada adoption indicates that lowest development cost still is the major award factor on DoD contracts, and that defense contractors perceive the DoD as unwilling to trade lower life-cycle cost for greater development cost [49].

Early perceptions: In the face of criticisms of early, and thus immature, Ada implementations, there has been little advertising of successful Ada efforts, such as those described in [87, 94] or the *Experience* track of the TRI-Ada conferences [108, 97, 38], and little concerted effort to gather, analyze, and distribute objective data about the economic impact of Ada on the software engineering discipline. The early bad press has left a legacy because of weaknesses of early implementations and the experiences of early Ada projects.

Language issues: Real and perceived language limitations have hampered the adoption of Ada. The Ada Joint Program Office (AJPO) has emphasized a strict validation process that has yielded hundreds of validated compilers. Great progress has been made in Ada compiler technology, including the development of optimizing compilers for many processors. Clearly, the image of Ada implementations having poor performance and quality is much outdated; projects should evaluate Ada implementations in light of their specific requirements.

Integration with computer-aided software engineering (CASE) tools: Current CASE tools are poorly integrated with most Ada implementations. Integrated environments that support some of the software engineering discipline encouraged by Ada are continuing to be developed.

See: Sections 3.2, 4.2.2.

Question: Who manages Ada?

Answer: The Ada language effort is managed by the Ada Joint Program Office (AJPO), a DoD office whose responsibilities include:

- maintaining Ada as military, Federal Information Processing Standard (FIPS), ANSI, and ISO standards, among others,
- developing compiler validation[4] procedures and guidelines,
- certifying facilities that perform validation of Ada compilers,
- coordinating the development of evaluation suites for Ada compilers, and
- coordinating DoD Ada training and education activities.

Stability of the language is an important consideration. Unlike many other languages where dialects and multiple versions have emerged, consistency over an extended period of time has been stressed. Subsets and supersets are discouraged, and compiler validation is used to ensure compliance to the language standard. The Ada 9X Project Office is managing the efforts to revise the Ada language standard (ANSI/MIL-STD-1815A) and to coordinate this revision with the international community to ensure Ada 9X adoption as an ISO standard.

See: Appendix B.3.

Question: What is the Ada 9X Project? What will its impact be?

Answer: The Ada language standard, ANSI/MIL-STD-1815A, was published in 1983. In 1988, it was recommended that the Ada standard be revised.[5] The Ada 9X Project[6] is the effort to revise the standard for the Ada programming language, obtain adoption of the revised standard, and effect a smooth transition from Ada 83 to Ada 9X.

The revisions leading to Ada 9X are intended to include only those changes that improve the usability of the language and minimize the disruptive effects of changing the standard. This is critical, as a significant infrastructure and investment exist in Ada 83. Ada 9X is planned to be mostly upwardly compatible with Ada 83 (i.e., most Ada 83 programs will compile and run under Ada 9X compilers).

The Ada 9X standard will incorporate improvements in object-oriented programming, programming-in-the-large, and real-time capabilities. ANSI approval of the Ada 9X standard is expected in 1994. DoD and National Institute of Standards and Technology (NIST) adoption of the ANSI Ada 9X standard will follow, and ISO approval will take at least two years after ANSI approval, due to voting procedures. In addition, the Ada 9X Project is taking several steps to hasten the availability of usable Ada 9X tools, including a change in the concept of the Ada language standard in terms of a *core* language and several *annexes*, which provide extended features for specific application areas.

---

[4]See Sections 2.3 and 6.2.1 for an explanation of compiler validation.

[5]Standards organizations (such as ISO or ANSI) require that all standards be periodically reaffirmed, revised, or allowed to lapse. This is not unique to Ada; all standards (including standardized languages) must undergo this process periodically.

[6]For more information about the Ada 9X Project, see Section 9. For contact information on the Ada 9X Project Office, see Appendix B.3.

For the program manager, some Ada 9X issues will be especially relevant (e.g., which Ada standard to apply, when to transition the program to Ada 9X or to continue doing upgrades using Ada 83). Use of Ada 83 for some projects is anticipated to continue well into the next decade.

See: Section 9, Appendix B.3.

2.2. Costs

The following questions are covered in this section:

- Will Ada save money?
- What impact will Ada have on hardware requirements?

Question: Will Ada save money?

Answer: Software development and maintenance organizations that use a single, modern, high-order language will save money. Individual programs that have not already adopted Ada will incur some one-time start-up costs for software tools, training, and development hardware (although in some cases, these costs may be shared by several programs). However, in the long run, the use of Ada is expected to reduce:

- individual program development and maintenance costs because of better software development methods, and
- overall costs through reduced post-deployment software support (PDSS) costs by using a single language for many programs.

See: Sections 3.1, 3.3, 5, 6.2, 6.3.

Question: What impact will Ada have on hardware requirements?

Answer: This question requires an examination of both host (or development) computer resources and target computer resources.

- **Host computer**: Ada compilers require somewhat more host computing resources than do compilers for most other languages, in part because Ada compilers detect many more programming errors earlier in the development process (thereby saving time during software integration) and provide greater levels of support for building large software systems than do other languages.
- **Target computer**: Ada compilers have been demonstrated to be as efficient as compilers for other languages. It is quite reasonable today to expect that Ada compilers will generate efficient code for resource-constrained machines. Of course, as with selecting any programming language, candidate compilers should be evaluated to ensure that they satisfy the needed code efficiency; not every vendor has emphasized target code efficiency.

See: Sections 6.2.2, 6.2.2.1, 6.2.2.2, 7.2, 7.3.

2.3. Technical Issues

The following questions are covered in this section:

- What is the current status of Ada compilers? Which compilers are most appropriate for a program?
- What is validation? How does the requirement to use only validated compilers affect a program that must modify its compiler?
- What is the purpose of an Ada program design language (PDL)?
- Can Ada be used on MIL-STD-1750A [73] computers? On the Motorola 68000 family? On reduced instruction set computers (RISC) processors or 32-bit processors? Are any of these processors more appropriate to use with Ada?
- Should every processor in a system be programmed in Ada?
- Can Ada be used in distributed applications?
- What is runtime software and why are runtime issues important?
- What is software portability? How can portability be maximized?
- Can Ada be mixed with other languages?
- Are there risks in automatically translating existing systems to Ada?
- Can commercial off-the-shelf (COTS) software, database management packages, and fourth-generation languages still be used with Ada?
- Can Ada help with software reuse?
- Is Ada an object-oriented programming language? Can Ada be used to implement an object-oriented design?
- Can cost and schedule risk be reduced by using C++ instead of Ada?

Question: What is the current status of Ada compilers? Which compilers are most appropriate for a program?

Answer: Robust Ada compilers are now available for a variety of processors, especially the more common ones and those used in mission-critical environments. Execution time appears to be comparable to other languages, although runtime checks (which are important in many safety-critical or fault tolerant applications) do add to execution time. A variety of utility programs, CASE tools, and training aids are available to support the Ada development environment.

Ada compilers are suitable now for applications that run on general-purpose computers with no severe memory or time-critical performance constraints. For performance and resource-constrained applications (avionics, fire control, etc.), numerous Ada compilers and runtime systems exist and have matured. Production quality compilers exist today. Today, the availability of a robust compiler for any specific host-target pair is more likely affected by market considerations than technical ones. That is, compiler vendors will support those products for which there is sufficient demand.

Each program should evaluate the quality and maturity of compilers based on their criteria, considering such factors as compile-time efficiency, object-code efficiency, additional compiler services, and support for embedded system requirements. Project-specific benchmarks, addressing project-unique requirements, provide a basis for evaluating compilers [226]. The Ada Compiler Evaluation Capability (ACEC) [100, 153, 154, 155] provides a standard method for evaluating Ada compilers.[7] Another method of compiler evaluation is embodied in the Ada Evaluation System (AES), developed in the United Kingdom [162]. These two benchmark suites are currently being merged. Also in use are the

[7]The first release of the ACEC occurred in August 1988. The current ACEC (Version V3) was released in August 1992.

Performance Issues Working Group (PIWG) benchmarks [187], which test performance characteristics of the software, and the Hartstone benchmarks [93, 92], which test a system's ability to handle hard real-time applications.

See: Sections 6.2.2, 7.1, 7.2, 7.3, 7.4, Appendix B.1, B.3.

Question: What is validation? How does the requirement to use only validated compilers affect a program that must modify its compiler?

Answer: Validation is a process developed by the AJPO to test the compliance of an Ada compiler with the language definition. To achieve validation, an Ada compiler must pass the Ada Compiler Validation Capability (ACVC) standard test suite. The validation process could be viewed as an *initial* test of a compiler; however, validation does *not* address performance areas such as compiler speed or efficiency of the generated code, so validation does *not* imply that a compiler is suitable (in terms of quality or features) for use by a specific program. According to the current validation guidelines [3], changes can be made to a compiler as long as they do not change the language. The validation tests should be run against a customized compiler to ensure there are no deviations from the standard.

See: Section 6.2.1, Appendix B.3.

Question: What is the purpose of an Ada program design language (PDL)?

Answer: A PDL is a formal notation used to capture design decisions. Ada's features for structuring and organizing programs are also usable during the design process to describe software architecture (the structure of a system), its interfaces, and its overall behavior, as well as to document that design. A design language should be used as a part of an overall software methodology. When placing an Ada PDL on contract, attention should be paid to how well it is integrated into the developer's methodology. Many ongoing procurements have called for the use of an Ada PDL, and DoD Instruction 5000.2[8] [79] suggests the use of a PDL.

Some advantages of an Ada PDL are:

- Using an Ada PDL is a good transition strategy — the designers work with and think in Ada terms earlier in the development, thereby potentially easing the transition from design to code.
- An Ada PDL should be compilable Ada (conforming to the language standard); then it can be checked by an Ada compiler on the host machine for consistency and completeness. An Ada PDL also provides a description of a design, useful not only to reviewers and others, but which can also be checked for interface errors.
- When made a part of the documentation requirements for software design documents, an Ada PDL can also be an indicator of the status and quality of a contractor's design efforts early in the development.

Some disadvantages of an Ada PDL are:

- Ada PDL has sometimes been applied in cases where the eventual implementation has been in a language other than Ada. While this is technically feasible, it has been difficult to carry out in practice, and should perhaps be avoided unless the designers can demonstrate an adequate understanding of modern software engineering principles and implementation tradeoffs.

[8]DoD Instruction 5000.2 [79] calls for the use of capable software processes and practices, including the "use of automated tools, such as computer aided software engineering (CASE) tools or formal manual techniques such as program design language and structured flowcharts." For more information about applying formal methods to Ada efforts, see [156, 188].

- Some aspects of design (e.g., performance requirements, traceability to requirements documents) are not readily captured in an Ada PDL. Structured comments known as annotations can be used to capture some of this information so that the resulting design is still compilable. Other aspects of a design (e.g., the overall design structure) are better represented with other notations [129].

There is no standard Ada PDL; however, several PDL processing tools are available, and there is an IEEE standard recommended practice for Ada as a PDL [132].

Question: Can Ada be used on MIL-STD-1750A [73] computers? On the Motorola 68000 family? On reduced instruction set computers (RISC) processors or 32-bit processors? Are any of these processors more appropriate to use with Ada?

Answer: Many existing compilers are mature for these processors. However, the mere existence of a compiler is not indicative of its quality or maturity. The quality of the code generated by the compiler (how well the application software will run) is somewhat dependent on the capabilities of the target hardware. While Ada does run on the processors mentioned above, Ada (and all modern programming languages) is best supported on processors with large memory capability and instructions for conveniently accessing that storage.

See: Sections 6.2.2, 7.2, 7.3.

Question: Should every processor in a system be programmed in Ada?

Answer: Certain situations require the use of special processor technology that is incompatible with any general-purpose language, including Ada. Some processors have highly specialized functions (e.g., a Fast Fourier Transform chip). Often these processors cannot in any practical sense support all the functions typically required by a high-order language. Other small processors are often used to provide limited functionality in a given situation (e.g., processors with limited address space and register set used for built-in test functions within an electronics assembly or as an input-output processor for a control panel). It may not always be appropriate to use Ada or any other general-purpose, high-order language on such processors because the expense of building the necessary support tools is expected to be high, and the advantages inherent in these specialized processors may be lost by using languages for which the hardware was not intended. However, since it is likely that these processors will be used in systems in conjunction with general-purpose processors whose software has been developed in Ada, the interfaces and communication between the processors are important for system engineering attention.

See: Sections 7.2, 7.3.

Question: Can Ada be used in distributed applications?

Answer: Some forms of distributed applications, especially loosely coupled distributed systems, can be implemented using Ada today, as was previously done with other languages, such as JOVIAL and FORTRAN. With the introduction of software technology currently in development, more sophisticated methods that take advantage of Ada’s tasking features should become available. Ada 9X is planned to provide increased support for distributed applications.

See: Sections 7.4, 7.5, 9.

Question: What is runtime software and why are runtime issues important?

Answer: Runtime software provides the additional supporting functions required for executing Ada programs on a specified target computer. Each target computer and each language places requirements on the runtime software. Ada runtime software is responsible for functions such as scheduling, parameter passing, storage allocation, and some kinds of error handling (much like the custom "executive software" written for applications developed in other languages, such as CMS-2, JOVIAL, and FORTRAN).

For further detailed discussion of runtime systems, see [224, 16, 17, 18, 89, 90, 91, 28, 10, 25, 26].

See: Sections 6.2.2.2, 7.4.

Question: What is software portability? How can portability be maximized?

Answer: Software portability describes the extent to which computer software can be moved without source-level modification between different computer systems. Portability has traditionally been a significant problem due to the proliferation of programming languages and their dialects. (FORTRAN is a classic example of a programming language with many dialects.) While some modification will almost always be required when moving software from system to system, Ada compares favorably to other languages because of:

- enforcement of the standard via the requirement for compiler validation, and
- language features (packages and private types) that allow software developers to isolate machine-dependent software. Through use of these features, the scope of required modifications is localized and chances of errors diminished.

Of course, Ada software must be properly designed to maximize portability; portability must be considered a design objective.

See: Sections 8.1.1, 8.1.3, 6.2.1, 6.2.2.4.

Question: Can Ada be mixed with other languages?

Answer: Mixing languages is possible and can result in productivity enhancements. Various projects have accomplished this successfully. An example of mixing languages is calling the FORTRAN International Mathematical and Statistical Library (IMSL) from Ada.

However, program managers should realize that if multiple language support is required, then potential Ada compilers must be evaluated with respect to their capability to mix languages. There are various levels of integration that can be achieved when interfacing Ada with other languages, existing systems, or commercial off-the-shelf (COTS) software. In some instances, the desired integration can be obtained, but will be unique to the particular implementation or not easily portable to another implementation.

See: Sections 6.2.2, 8.2, 8.4.1.

Question: Are there risks in automatically translating existing systems to Ada?

Answer: There are significant risks in attempting automatic translation from any programming language to another, and this strategy is generally not recommended as a means of achieving Ada-based software. A thorough analysis of technical, cost, and life-cycle issues should be accomplished before committing to an automatic translation approach. Re-engineering existing systems may be the most effective way to achieve the desired results [46].

See: Section 8.4.4.

Question: Can commercial off-the-shelf (COTS) software, database management packages, and fourth-generation languages (4GL) still be used with Ada?

Answer: Ada applications can effectively use whatever resources (i.e., operating systems or communications services, database management systems, graphics packages or user interface management systems, or interfaces to COTS software or other existing systems) are needed to produce a complete system. Ada interfaces to many other systems and standards are available [133]. For example, Ada-SQL interfaces, or bindings, exist. New implementations of interfaces (or bindings) are continuing to emerge. Since much attention is presently being paid to successful integration of Ada software in overall systems environments, program managers should determine if bindings exist before beginning additional development.

Fourth-generation languages, when used in their proper domain, make significant increases in productivity possible. Some applications may indeed be suitable for fourth-generation language approaches. However, 4GLs typically provide productivity gains only through reductions in coding efforts and can prove very costly if used to satisfy requirements even slightly outside their domain of applicability. Service-specific policies identify the requirements for waivers or exceptions when using such technologies.

See: Sections 8.2, 8.3.4, 3.2.3.

Question: Can Ada help with software reuse?

Answer: Many functions in new software systems are similar, if not identical, to those in previously developed systems. Reusing existing system requirements, design, and code and applying them to new development efforts or porting existing software to new hardware configurations has the potential to significantly reduce development costs and to produce more reliable systems. Ada's standardization and some of its special features mean that Ada is especially suitable for promoting software portability and reuse. These language features are available to facilitate portability and reuse of both software code and software architectures.

See: Section 8.1.

Question: Is Ada an object-oriented programming language? Can Ada be used to implement an object-oriented design?

Answer: Object-oriented methodologies can be used in specifying requirements, designing, and implementing systems using Ada [177, 203, 120, 2, 20, 150].

Ada 83 has been referred to as an "object-based" language, but it is not a complete "object-oriented" programming language because it does not support full inheritance and runtime polymorphism.[9] Ada 9X will extend the "object-based" features of Ada 83 (such as abstract data types and derived types) to incorporate the object-oriented *class* concept[10] and will be fully supportive of object orientation.

Object-oriented design (OOD) typically refers to a variety of design strategies (e.g., Booch) that Ada supports well. Object-oriented programming (OOP) refers to a programming paradigm involving inheritance of classes, which Ada 83 does not support well [201], but Ada 9X will support better. Commercial products, such as language preprocessors, are available to give Ada full object-oriented

[9]See [223] for a discussion of the distinction between "object-based" languages and "object-oriented" languages. *Inheritance* is a means for incrementally building new abstractions from existing ones by "inheriting" their properties — without disturbing the original abstractions' implementation or other inherited uses [7]. *Polymorphism* is a means of factoring out differences among a collection of abstractions such that programs may be written in terms of their common properties [7].

[10]For a more detailed summary of these enhancements, see [215].

capabilities. There is nothing in the language to prevent software from being implemented in Ada based on object-oriented requirements analysis (OORA) or object-oriented design.

See: Section 9.

Question: Can cost and schedule risk be reduced by using C++ instead of Ada?

Answer: Software for large, complex applications must be engineered. It was precisely these kinds of large applications that Ada was designed for. In general, DoD policy requires that Ada be used for defense software, and a waiver must be approved to use another language. A waiver request must address the justification for the request, life-cycle cost analysis, and risk analysis addressing technical performance and schedule impact.

Although it is apparent that C/C++ has gained widespread market acceptance in recent years, popularity and market acceptance are not always indicators of technical superiority, nor are they reasons for choosing one language over another for a particular application. Language comparisons are often futile efforts because they fail to consider the context (i.e., application domain and requirements, hardware and software tradeoffs, projected system life cycle, and associated development and support costs) in which the language will be used.

The Air Force conducted a study to examine the circumstances in which the use of C++ might be justified for DoD software development [81]. Each portion of this study reached the same conclusion: "there is no compelling reason to waive the Ada requirement" to use C++. Particularly noted were Ada's suitability for large systems, its safety (ability to detect errors), and its maturity (through standardization and compiler validation) as compared to C++.

Until Ada 9X is available, C++ may provide some advantages for object-oriented programming. For small systems or prototypes where existing class libraries can be used to build systems from components, it may be possible to justify C++ on a cost and schedule basis, just as 4GLs can be justified for certain applications. However, waivers will generally not be justified by lower development cost or lower risk for large, long-lived, or safety-critical DoD applications, as life-cycle issues are the drivers for these systems.

See: Section 4.2.2.

2.4. Program Management

The following questions are covered in this section:

- What Ada training sources exist? What topics should be emphasized?
- What is the best strategy for an organization that has little or no experience developing systems in Ada?
- How should those proposing Ada be evaluated during the source selection process?
- Will changes be required in software management methods (e.g., design walkthroughs and code inspections)?
- Compared to other languages, how many source lines of code (SLOC) will Ada require for equivalent functions?
- Will Ada affect cost-estimating models?
- How productive can Ada software developers be?
- Can Ada be used with DoD software development standards?
- What effect does Ada have on configuration management (CM)?

Question: What Ada training sources exist? What topics should be emphasized?

Answer: Ada training courses are available from a wide variety of government, academic, and commercial sources. To obtain the full benefits of using Ada, software engineering concepts should be taught as an integral part of the overall training program, since many software developers do not have the appropriate background in this area. Additionally, hands-on design and programming exercises are essential for software developers. The training needs of managers and other personnel (e.g., application specialists, lead designers, software engineers, and testing personnel) should be considered in planning a transition to Ada.

See: Section 5, Appendix B.3.

Question: What is the best strategy for an organization that has little or no experience developing systems in Ada?

Answer: There are many approaches (for example, addressing compiler and tool acquisition and system engineering concerns) that are detailed in other chapters of this handbook. Perhaps most important, given this situation, is to develop an internal cadre of skilled Ada personnel.

See: Sections 4.1.2, 4.3, 5, Appendix B.

Question: How should those proposing Ada be evaluated during the source selection process?

Answer: In general, contractor evaluation should be done in a manner similar to that used for other software developments [40, 80, 47, 82, 62, 51, 22 (Section 10)]. Expertise in Ada and software engineering should be a major discriminator in contract awards. A proven record of application-domain expertise and meeting cost, schedule, and quality requirements is important. Additionally, an assessment of Ada capabilities is necessary in terms of people, software tools, training, methodology, management, ongoing internal research and development efforts, and previous developments; program risk increases if a contractor attempts a large Ada development without any Ada experience. Finally, Ada-knowledgeable staff can be consulted as part of the evaluation process.

Question: Will changes be required in software management methods (e.g., design walkthroughs and code inspections)?

Answer: While software management techniques such as design walkthroughs and code inspections still apply, experience on Ada projects indicates that the levels of effort and time associated with the various phases of the software life cycle have changed with the use of Ada. Experience to date indicates the need to spend more time in the definition and design phases and less time in testing and integration than was spent in previous efforts. Increased emphasis on the design phase will lessen the time involved in testing and integration, since many errors will be identified early in the development [24, 29], and may also decrease the time required in the coding phases [175, 193]. Some rules of thumb to use regarding levels of effort are:

Phase	**Other Languages**	**Ada**
Design	40%	50% +
Code	20%	15%-30%
Test and Integration	40%	20%-35%

See: Section 4.1.1.2.

Question: Compared to other languages, how many source lines of code (SLOC) will Ada require for equivalent functions?

Answer: Software measures are useful for gaining insight into a software program. Software size, often measured in terms of source lines of code (SLOC), is one such measure.[11] There are many different views of what a source line of code is and how to use the computed values [104]. Although the validity of SLOC has been questioned, and numerous definitions and counting schemes exist, an understanding of the issues and concerns is important, since SLOC is sometimes used as an indicator of system complexity, system hardware requirements (processors and memory), and as a basis for computing system cost.

Any estimates comparing Ada and other languages are *inexact* since factors such as individual capabilities, application characteristics, domain complexity, design and coding techniques, and language feature use will exert strong influences. The counting technique used will also make a substantial difference [104]. A number of sources have identified definitions of SLOC and counting methods; these include [182, 230, 194, 176]. Regardless of the definition used, it is imperative that lines of code be estimated and counted in a consistent fashion.

Following are *general* comparisons between Ada and other languages using the number of source statements per function point[12] [139]; it should be noted, however, that as software reuse and generics become more common, the number of *original* lines (that have to be written from scratch) may well decrease.

Language	Source Statements per Function Point
Assembly	320
C	128
FORTRAN 77	105
ANSI COBOL 74	105
Pascal	91
Ada	71
Fourth generation language (4GL)	20

Table 2-1: General Comparison of Ada and Other Languages

Jones [139] notes that these are general trends and that great variability caused by many factors, including differences in programming styles, can influence these comparisons. An example of this sort of difficulty in making language comparisons is seen in comparing parallel Ada and FORTRAN efforts [168]. The resulting two systems differed in several measures of SLOC, depending on what was counted. A major difference between the smaller FORTRAN and the larger Ada program is that a substantial amount (at least 30,000 SLOC) of additional functionality was built into the Ada user interface, and that the coding styles for Ada were not the same as the FORTRAN styles (i.e., Ada used longer prologues and more commenting). Again, much of this comes down to how measurements are made, what is being measured, and the variability of the people and processes used.

[11]A description of several common software development measures, including software size, can be found in [197].

[12]The *function point* metric was developed as a relative measure of the amount of user function delivered to the user or customer independent of the particular technology, language, or design approaches used [9]. The function point measure is claimed to be more useful than SLOC as a prediction of work effort because this measure can be estimated from the basic requirements for a program early in the development cycle.

Managers are urged to gather and use data as part of a meaningful software measurement process [30, 197]. In this way, meaningful numbers for both lines of code and productivity can be developed.

Question: Will Ada affect cost-estimating models?

Answer: Several of the major cost-estimating models (REVIC, COCOMO, SOFTCOST, etc.) have been revised to reflect experience with Ada [34, 142, 193, 143]. Experience from initial Ada programs indicates that several cost factors will change, including productivity, definition of lines of code, and levels of effort associated with various phases of development [29, 191]. Cost models need to be recalibrated to reflect differences as a result of an organization's use of Ada [193].

See: Section 4.1.1.2.

Question: How productive can Ada software developers be?

Answer: While only a few definitive Ada productivity studies have been conducted [193, 168, 29, 107, 191], Ada software development efforts generally report that overall productivity is higher than for other languages. Improvements (after an organization's initial Ada effort) as high as a factor of two have been reported informally [168]. Studies have concluded that the productivity on a third-time Ada project result in a slightly lower cost than that of an equivalent FORTRAN project and that these projects show significant decreases in error rates [8, p. 6-2]. Once an organization has adopted Ada, average productivity gains of 20 percent have been reported [193]. It is difficult to tell how much of these productivity gains is a direct result of using Ada, and how much is attributable to the use of software engineering methods and techniques, improved software processes, increased levels of reuse, and domain-specific architectures.

However, program managers should be aware that it is also *not* unusual for productivity at the beginning of a program or with first use of a technology such as Ada to be low due to a learning curve [193, 145]. (A good training program is essential in reducing the learning curve.) Additionally, cautious use of reported data is recommended since productivity will vary due to:

- the experience and capabilities of people,
- the size and complexity of the system under development,
- volatility of requirements specifications,
- maturity and sophistication of the software development environment, and
- documentation requirements.

Question: Can Ada be used with DoD software development standards?

Answer: Ada can be used with DoD software development standards [74, 75, 72]; however, as with any DoD standard, tailoring may be required for the particular needs of any program. Discussion of some DoD-STD-2167 and DoD-STD-2167A related issues may be found in [221, 103, 116]. Significant work in this area has also been accomplished by the ACM SIGAda Software Development Standards and Ada Working Group (SDSAWG). Several documents, such as DoD-HDBK-287 [76] and program-specific guidance [148], have been developed to provide guidance in tailoring DoD-STD-2167A.

See: Section 4.1.1.1, Appendix B.1.

Question: What effect does Ada have on configuration management (CM)?

Answer: Disciplined CM practices are a prerequisite to effectively managing Ada software development. CM for small and moderately sized Ada systems (less than 100,000 lines), as supported by traditional file oriented tools,[13] is largely indistinguishable from CM for systems written in other programming languages.

[13]Examples would be DEC's MMS and CMS or UNIX sccs and make.

However, the CM requirements for large software systems tend to be much more complex than for smaller systems, independent of programming language. This is especially true for Ada, which is intended for the design and development of large systems. Among the issues that have to be addressed are multiple implementations of package bodies, and the growing number of interdependencies among objects as systems increase in size.

For large systems, proper system management will require support from the program library since the compiler is involved in supporting functions that previously had been managed by the environment support tools. Since CM for large Ada systems is an area for concern, the program manager should ensure that Ada program libraries provide support for automatic recompilation, and evaluate the level of integration between the Ada compiler and the project-selected CM tools. Ada does have an impact on CM practices because Ada code sequences are placed under configuration control much earlier in the development process than was true for other languages, primarily because of Ada's use as a PDL and its separate compilation features. Performing CM for Ada requires substantially more computing resources than for other languages because Ada enforces interface compatibility among independently compiled modules.

2.5. For More Information . . .

Many sources of information are listed in the appendices to this handbook. Two sources are included here for quick reference:

- **Ada Joint Program Office (AJPO)**: Contact Dr. John Solomond, Director, AJPO, The Pentagon, Washington, D.C., (703) 614-0208.
- **Ada Information Clearinghouse (AdaIC)**: Contact the AdaIC at (703) 685-1477, (800) AdaIC-11, FAX (703) 685-7019, or electronically via adainfo@ajpo.sei.cmu.edu or CompuServe 70312,3303.

See: Sections 5.5, Appendices A, B.

3. The Need for Ada

3.1. The Software Problem and Ada's Role

Software is increasingly being recognized as a serious problem in the development of mission-critical systems, regardless of whether these systems are embedded systems or information systems necessary for ongoing operations. Two difficult aspects of this software problem are proper development (i.e., on time, within budget, with required functionality) and recognition of the need to prepare for many years of ongoing software support and enhancements. Ada was designed to address these problems by:

- *reducing the number of programming languages needed so effort could be focused on making one language work well; and*
- *facilitating the use of modern software development methods, thereby reducing the costs and risks of software development, and facilitating improved maintenance during post-deployment support.*

Many DoD and industry studies have documented that software is assuming a greater share of system complexity and cost, and demand for quality software has been rising faster than the ability to produce it. About 40 percent of the functionality of the F-16 fighter depends on software as does about 80 percent of the functionality of the F-22 advanced tactical fighter [144]. Indeed, systems of hundreds of thousands, even millions of lines of code are becoming increasingly common, resulting in situations where computing systems are on the critical path to systems acquisition and among its leading problems.[14]

The growing importance of software, in terms of dollars or necessary capabilities, cannot be overstated. Some estimates have predicted that DoD software costs could rise as high as $50 billion by 2006 [218]. Col. Donald Carter, U.S. Air Force, former acting deputy undersecretary of Defense for Research and Advanced Technology, told the House Armed Services Subcommittee on Research and Development that "software is the human intelligence that is programmed into our systems. It allows advanced sensors to discriminate and track, navigation systems to follow prescribed routes, guidance systems to control trajectories, and communications systems to properly route thousands of messages. Software keeps track of the status of our forces, maintains intelligence information on enemy forces, and aids our commanders in deciding on target actions" [141].

Software engineering is the discipline in which quality (efficient, reliable, maintainable) software is produced within the constraints supplied by contractually specified acquisition, development, operational performance, and life-cycle support considerations. Software engineering practice today lags the ideal of a robust, mature engineering practice [219]. The DoD is aware of the software cost and technology insertion problems in the development and post-deployment support of software, and established a software initiative consisting of the Ada Program, the Software Technology for Adaptable, Reliable Systems (STARS) Program, and the Software Engineering Institute (SEI) to address the problem. The Ada

[14]To illustrate the magnitude of a delivered 1,000,000 (10^6) line application software system, assume that it is printed on 8.5" x 11" paper with 50 lines per page. The resulting listings would be a stack of more than 20,000 pages. Assuming that 500 sheets is about 1.5" thick, the stack would be approximately 5 feet high. Note that the above sizing omits any consideration of code to provide the software development environment, simulation software, and test software, and also omits documentation, contractually required data items, and presentation material [118].

language effort focuses programming development methods and tools on a single language that supports modern software engineering techniques. Ada’s role as the single, common, high-order programming language for defense systems [79] is a major step forward in addressing DoD software development problems. By late 1989, over 58 million lines of Ada code for defense and commercial applications were planned, in development, or being maintained [207].

3.2. Policies on the Use of Ada

Although Ada is an internationally recognized standard, there are specific policies that mandate its use. This section discusses the congressional mandate, DoD policy, and the policies of the DoD services and other agencies.

In March 1987, the International Standards Organization (ISO) approved Ada as an international standard, *ISO/8652-1987, Programming Languages — Ada*.[15] This ISO standard is identical to the DoD standard adopted for Ada [13] and provides a standard language definition that can be used throughout the DoD, industry, and the world.

3.2.1. Federal Legislation

Within the United States, policies mandating Ada’s use flow in part from federal legislation.

The U.S. Congress enacted legislation (in the FY91 DoD appropriations bill, Public Law 101-511, Section 8092) specifying the use of Ada for DoD software development:

> Notwithstanding any other provisions of law, after June 1, 1991, where cost effective, all Department of Defense software shall be written in the programming language Ada, in the absence of special exemption by an official designated by the Secretary of Defense.

This congressional action was motivated by a belief that "there are still too many other languages being used in the DoD, and thus the cost benefits of Ada are being substantially delayed" (House Report 101-822).

3.2.2. Department of Defense Policy

Department of Defense policies implement the federal legislation mandating the use of Ada.

The two major DoD statements of policy guidance regarding the use of Ada are found in:

1. DoD Instruction 5000.2 [1991] - *Defense Acquisition Management Policies and Procedures* [79]
2. DoD Directive 3405.1 [1987] - *Computer Programming Language Policy* [77]

DoD Instruction 5000.2 [79] requires that:

> Ada is the only programming language to be used in new defense systems and major software upgrades[16] of existing systems.

[15]Although the ISO standard was approved in 1987, the existing Ada language is called Ada 83, at least in the United States, because the ANSI standard [13] was approved in 1983.

[16]A major upgrade is defined as the redesign or addition of more than one-third of the software.

This policy establishing Ada as the *only* programming language included:

- Approval of ATLAS [131] for use in automatic test equipment.
- Approval to use languages other than Ada for deployment or software maintenance (but not for major system upgrades), if they were approved languages when used in engineering and manufacturing development.
- A preference (but not a requirement) for Ada in commercial, off-the-shelf (COTS) applications procured for use by the DoD.
- The requirement that only validated Ada *compilers* be used.

DoD Directive 3405.1 defines as Department of Defense (DoD) policy the use of "modern software concepts, advanced software technology, software life-cycle support tools, and standard programming languages." It defines DoD policy to transition to the use of Ada to limit the number of programming languages used within the Department of Defense. This policy also declares the DoD intent to "reduce software obsolescence and the cost of software maintenance through the use of approved programming languages and appropriate advanced software technology during all phases of the software life cycle."

This directive defines Ada as:

> "the single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system."

According to the directive, Ada will also be used for all other applications, except when the use of another DoD-approved high-order programming language (HOL) is shown to be more cost effective in a life-cycle cost analysis. In fact, the directive defines a hierarchical preference for the use of software technology to meet functional requirements, based on an analysis of life-cycle costs and impacts. These preferences are (in order of preference):

1. Off-the-shelf application packages and advanced software technology
2. Ada-based software and tools
3. Other approved standard HOLs

This directive defines a waiver mechanism for systems or subsystems that cannot comply with the programming language requirements of the policy. Each waiver request must address these factors:

- life-cycle cost analysis
- risk analysis, addressing both:
    - technical performance risks
    - schedule impact

3.2.3. Service Policies

DoD policies on Ada use are reflected in the policies of each of the services.

Each of the services (and their subunits) has developed implementing policies regarding the use of Ada within their service. These are the:

- Army *Implementation of the Ada Programming Language* [83]
- Navy *Interim Policy on Ada* [84]
- *Marine Corps Ada Implementation Plan* [85]
- Air Force *Interpretation of FY 1991 DoD Appropriations Act* [70]

3.2.4. Other Policies on Ada Use

Numerous governmental agencies have defined policies regarding Ada's use.

Ada has been a NATO standard for several years. And in the civilian sector, the Federal Aviation Administration, in its Software Procedures, has selected Ada as the single, common, high-order programming language used in the National Airspace System (NAS) [86]. Although there is currently no NASA-wide requirement to use Ada, the NASA Space Station *Freedom* Project is currently required to use Ada.

3.3. Ada Benefits

Ada's major advantage is in providing the means to achieve the benefits of modern software engineering methods.

Ada was designed to address increasingly serious problems in development and post-deployment support of mission-critical systems by:

- **Standardization**: Reduce the number of DoD programming languages (estimated in the early 1970s at between 300 and 1000).[17] Ada is a now a DoD, FIPS, ANSI, ISO, and NATO standard, among others.
- **Discipline**: Encourage the use of modern software development methods.

In a word, *cost* is why Ada was developed: to address increasing costs due to use of multiple languages and old languages and technologies. Ada was designed with three overriding concerns in mind: program reliability and maintenance, programming as a human activity, and efficiency [13]. By addressing these concerns, Ada offers potential solutions to many software development problems.

The following benefits become more important with increasing project size:

- **Reuse**: Ada's constructs and features are particularly well suited for the development of reusable components and subsystems. Ada's software modules can be made generic to maximize reusability.
- **Code portability**: Ada makes it easier to move source code between different computers with a minimum of changes (often referred to as machine independence). This can be a real issue in maintaining systems with long lifetimes.
- **People portability**: With language standardization, software personnel will be more able to move from project to project with significantly less need to learn a new language. (While Ada is standardized, differences among compiler and tool implementations still exist.) Additionally, with time, there should be a larger pool of people able to work on any given project.
- **Maintainability**: Ada can reduce costs to make modifications, enhancements, and upgrades in software.
- **Reliability**: Ada can reduce errors during development and maintenance activities by providing earlier identification of certain types of programming errors. Such capabilities will help increase system reliability.

[17] The DoD moved toward language and processor standardization when there were more than 300 different computer programming languages and more than 1000 dialects in use on DoD systems, as well as perhaps hundreds of target processors. This created an N $\times$ M (languages $\times$ processors) problem of enormous proportion, as the system development and post-deployment software support (PDSS) costs of this situation in terms of tools, people, training, and maintenance rapidly escalated.

- **Common basis for tools and methodologies**: Ada is serving as a focal point for investing in and developing high-quality tools and methodologies. Previously, tool and methodology development efforts were diffused across many languages, resulting in less capable and lower quality tools.
- **Managing complexity and modularity**: Complexity has become a major factor limiting progress in the software industry. Ada helps manage complexity by explicitly supporting the process of dividing a problem into well-understood pieces whose interfaces and interactions with other software and hardware components are well defined. This support is critical because, in general, the approaches, techniques, and languages that have historically been used to build software systems do not address larger problems very well. Ada is therefore becoming a necessity for large systems to function correctly and reliably. It a tool that was designed to support the complexity and versatility required.
- **Management visibility, more emphasis on a system view**: Ada's features encourage a more disciplined approach to defining interfaces and making design decisions; hence, managers have the opportunity to better understand and control their progress against schedule.
- **Productivity**: After experiencing a productivity loss during initial Ada projects, organizations that have adopted Ada report average productivity gains of 20% after Ada adoption [193].

While the potential advantages of using Ada are great, its mere presence and supporting technology do not guarantee success. Achieving Ada's benefits means investing in training, tool development, pilot projects, and development standards to achieve the resulting benefits over the entire system life cycle.[18]

Will Ada save the DoD money? Savings could be made by the DoD overall in:

- Anticipated cost reductions due to the aggregate use of a single modern, high-order language.
- Anticipated benefits through standardization and more up-to-date tools/management procedures.
- Possible amortized costs due to software reuse.

An individual DoD program manager looking only at development costs may recognize that individual programs may incur some one-time start-up costs; for example, in acquiring software tools, training, and development hardware. However, these costs may possibly be amortized across programs through standardization. Taking a life-cycle perspective, a DoD program manager should expect the use of Ada to reduce individual program development and maintenance costs in the long run. This perspective is crucial, as post-deployment software support (PDSS) is perhaps the most rapidly growing segment of the DoD software workload [207].

Ada's importance in supporting software engineering has also been recognized outside the DoD. Ada has achieved market acceptance in certain civilian application domains (e.g., air traffic control, commercial avionics, and space applications); it is also being used in a selected number of other commercial developments. Commitments to use Ada in commercial developments have been made by companies such as IBM, Weirton Steel, Shell Oil, Wells Fargo Bank, Motorola, and Boeing Commercial Airplane Company in the United States, Nippon Telephone and Telegraph in Japan, and Thomson-CSF, BOFORS, Philips, Nokia, and Volvo in Europe. These companies have chosen Ada because they are convinced that the technology is mature enough for their applications (which include communications,

[18]The system life cycle is the span of time over which a system is in existence, starting with its conception and ending with its last use. System life cycles are usually divided into conceptual, developmental, production, and operational phases, and span many years.

manufacturing, air traffic control, defense systems, banking, and teleprocessing) and the advantages are worth pursuing. In fact, foreign usage of Ada is growing faster than domestic usage [136]. Ada is not just a DoD language. Its success is important to private companies as well, and the DoD can share in the benefits of commercial interest and development.

However, it takes management action, not just mandates, to reap the benefits of Ada. Managers have found that through careful management attention, they can go beyond the minimal rewards of mere policy compliance and reap benefits from using Ada. The following chapter addresses these management issues in light of adopting Ada.

4. Management

The transition to Ada should be accompanied by adoption of other software engineering technologies, methods, and tools. Consequently, the costs of transitioning to Ada may appear to be higher than the costs of transitioning to other languages in the past.

Size (in lines of code) and complexity (system functionality implemented) of many software systems has increased dramatically, making it a pacing item. These increases in size and complexity, regardless of language used*, deserve management attention.*

The resulting management strategy must incorporate considerations for software process, personnel, and technologies that are appropriate for organizations adopting Ada and software engineering techniques.

Adoption of Ada may require adjustments in management strategy.[19] Ada has been called an "enabling technology" — a technology that facilitates or *enables* the use of other technologies. What Ada has enabled is the introduction, in a more systematic way, of many of the fundamentals and discipline of software engineering.

Not only must managers adopt strategies for dealing with Ada, but they often must also simultaneously incorporate strategies for improving the end result of the software engineering processes. For such strategies to be successfully applied, management should be aware of, and plan for, the possibility that initial projects using the new technology may be more expensive than later projects and may even be more expensive than with the old technology[20] [193]. Although these strategies may address general issues relevant to the introduction of any new technology or the improvement of software engineering practice, we will look at them in the context of Ada.

Ada, even though it was standardized in 1983, may still be a "new" technology in many organizations or in organizations where Ada is used in other parts of the organization. Adopting Ada and new software engineering practices presents a number of choices that must be made in the context of the organization.

4.1. Management Issues

Management issues in adopting Ada and managing Ada projects reflect Ada's impact on the software process (tasks and activities, people, and tools and technology).

Many oft-revisited software-related problems relate to management and *not* the technology itself.[21] Ada is part of a larger strategy for improving software production and support capabilities; however, Ada itself is not a silver bullet. One of its major strengths is that it supports the use of sound software engineering

[19]This section is not intended to be an overview of managing software acquisition or software development. Other sources (such as [160, 198]) already provide a broad coverage of such topics. This section is intended to deal specifically with the impacts of Ada and Ada adoption on management. The general issue of using and managing Ada efforts has been discussed for some time (for example, [173]). But it is important to note that many of the concerns cited in earlier reports, such as [173], do not have the import that they had a few years ago — the compilers have matured and more is known about using Ada.

[20]Kemerer [145] discusses this phenomenon in the adoption of CASE tools.

[21]See also [68] and predecessor studies for further elaboration on this point.

practices; its successful use requires an understanding and appreciation of software engineering and project management. In comparing U.S. software strengths and weaknesses, it has been noted that, in general, problems are centered on managerial issues, and strengths are focused on tool and technical staff skills [139]. This is also true of Ada efforts, as program management problems encountered by managers of large Ada software developments are often not specific to the Ada language, but are related to difficulties in software engineering and program management [185].

Large software systems can be well engineered and well managed, and Ada has shown that it can help. Like any technology, Ada:

- has up-front costs
- requires educated and trained professionals
- yields real benefits when intelligently applied

In adopting Ada (or any new technology), managers must be aware that the technology is not only new, but has the potential for enabling changes and improvements in the organization and its software process; specifically in the tasks and activities performed (Section 4.1.1), the people who manage and develop software (Section 4.1.2), and the tools used (Section 4.1.3). These relationships are depicted in Figure 4-1 below.

4.1.1. Software Tasks and Activities

An organization's defined software process is the basis for managing its software tasks and activities.

Fundamental problems in software engineering today involve an inability to manage the ever-increasing complexity of software systems and the lack of a *disciplined engineering approach*.[22] A consequence of this is the need to understand normal corporate capability and capacity. This is true not only of defense-related software engineering. Similar problems are affecting the latest generations of personal computer (i.e., mass market) software [200]. These issues can occur whenever the size and complexity of the systems being built exceed the abilities of one or two people to have intellectual control. In essence, this is where software engineering *"in the large"* comes into play.

If Ada supports good software engineering practices (or is adopted with, or encourages the use of, such practices), part of the difficulty in quantifying the impact of the language is that Ada's impact is intertwined with the improvements brought about by the use of these modern practices. Transition and start-up costs are also intermingled with the ongoing costs of doing business (at least for the first few Ada projects). Many firms and acquisition organizations have an inadequate collection of prior cost, quality, or productivity data to compare against actual results from new projects using Ada.

Characterizations of the state of the practice of software engineering have indicated that many software organizations are operating at an immature level of software process maturity [147, 126]. According to the SEI five-level process maturity model, current software engineering practice is largely at the initial

[22]In thinking about a *disciplined engineering approach*, a comparison to bridge-building is a good one. That activity has well-determined techniques for soil characteristics analysis, cost estimation procedures, known formulas, and generally assumed reliability of both process and product. A similar domain knowledge base generally does not yet universally exist for engineering software systems.

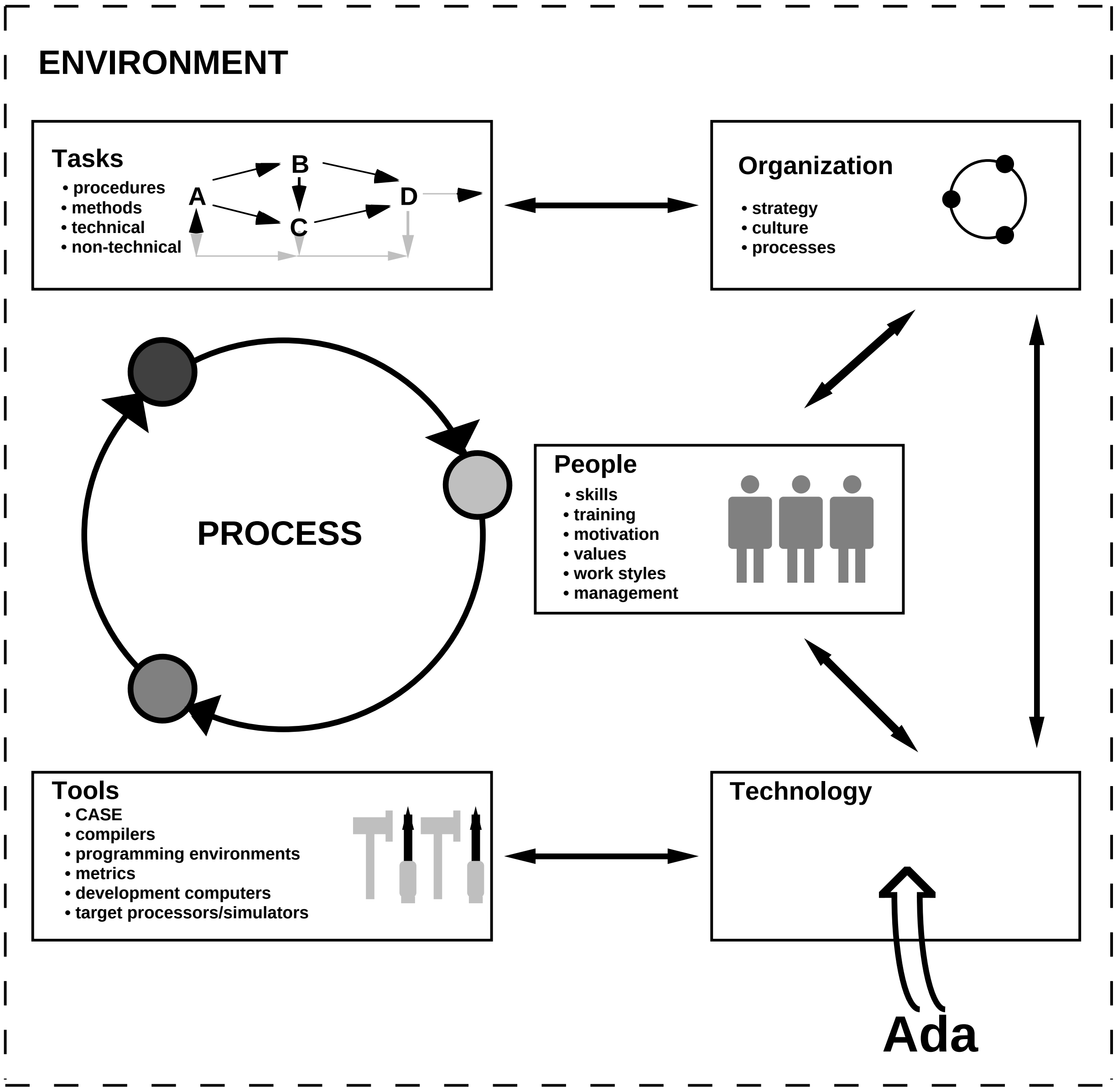

Figure 4-1: Potential Influences of Ada Adoption

(level 1, or lowest) level of maturity. Only a small number of organizations have repeatable (level 2) or defined (level 3) processes [147, 126, 229, 127].

Nearly all level 1 software organizations urgently need to improve their management systems for controlling their software engineering efforts [126]. These organizations need improvements in conducting project reviews, measuring and examining key indicators of engineering progress, and applying basic management methods and techniques. Level 2 organizations were found to typically have their costs and schedules under reasonable control, but not to have orderly methods in place for monitoring, controlling, and improving the quality of their software or the software processes that they use [126].

However, many of these organizations have begun steps toward long-term improvement; Hughes has reported an almost ten-fold return on investment from these efforts [127]. One such step is the establishment of a software engineering process group (SEPG), a group of software professionals specifically chartered to focus on software process improvement.[23] In addition to focusing on software process improvement [146], this group can facilitate the mapping of generic improvements in technology, processes, tools, methods, or environments into the specific context of the organization. Many organizations adopting Ada have used a similar group to develop and spread Ada expertise throughout the organization.[24]

One of Ada's design goals was to encourage the use of modern software development methods, thereby reducing the costs and risks of software development and facilitating improved maintenance throughout the complete software life cycle. An organization's successful use of the Ada programming language may enable that organization to make other improvements in its application of software engineering principles and management of the software engineering process while implementing Ada. Thus, Ada may serve as an enabling technology for broader organizational improvement.[25]

4.1.1.1. Software Development Process

Ada supports the development process, not the other way around. In fact, Ada should be a small part of the overall process. Using Ada should not obscure the fact that there is a specific problem to be solved, as systems are created to do a function, not solely as an exercise in applying a specific language.

While serving as a new technology that may enable improvements, Ada must become an integral part of an organization's software development process. This is true at all levels of consideration, from the life-cycle management plan [79, Part 6, Section D] defining acquisition strategies, to the individual software development plans for developing software.

Existing standards [74, 75, 72] can be used with Ada. However, as with any DoD standard, tailoring may be required for the particular needs of any program. Discussion of some DoD-STD-2167 and DoD-STD-2167A related issues may be found in [221, 103, 116]. Significant work in this area has also been accomplished by the ACM SIGAda Software Development Standards and Ada Working Group (SDSAWG). DoD-HDBK-287 [76] and other documents, such as [148], have been developed to provide guidance in tailoring DoD-STD-2167A. Others [120, 2] have suggested tailorings appropriate for object-oriented techniques.

---

[23]See [110] for more information on establishing a software engineering process group and related software engineering process improvement functions.

[24]See [209, pp. 43-44] for a representative set of responsibilities for an Ada-focused support group.

[25]A broader discussion of such an improvement framework using Ada as an enabling technology can be found in [119]. Numerous lessons learned that are relevant to understanding the adoption of Ada [32, 208], introduction of related software engineering technologies [32, 124], or software process assessment as a starting point for continuous improvement [146] can provide insight when planning improvement efforts using Ada as an enabling technology.

DOD-STD-2167A specifies a set of activities that must take place during a software development project, and states that: 1) it is *not* intended to specify or discourage the use of any particular software development method, and 2) the contractor is responsible for selecting software development methods that best support the achievement of contract requirements. This standard can also be tailored to match the planned efforts. For example, strategic defense system (SDS) software development has addressed this issue in light of selected life-cycle approaches, and have developed a set of guidelines for tailoring DoD-STD-2167A for SDS software development [148]. Other efforts [130, 222] define the processes and practices that are necessary for successful software development. For example, IEEE STD 1074 emphasizes the software processes themselves, rather than the time-sequencing of those processes.

It is incumbent on each development organization, as a normal part of software development, to select and *follow* an appropriate life-cycle model. The most common problem in federal information technology efforts has been inadequate management of the development life cycle [112]. The software processes (such as those described in 2167A or the IEEE standard) and the activities that comprise those processes must be mapped into and managed as part of the selected life-cycle model, regardless of which model the selected model is based on.[26]

In performing this planning, compatibility with the developer's existing methods, tools, and capabilities needs to be considered, as well as the impact of Ada on the development process. Not only should the process model adopted by an organization be compatible with its needs in developing software, but the software engineering practices and methodologies used should support the adopted process model.

4.1.1.2. Ada's Impact on the Software Development Process

Ada alone will not solve all software management problems. But, Ada has been shown to have an impact on the software development process and on the resulting products.

Most of the problems encountered by Ada programs are management-related, not Ada-related, problems [176]. In addition, some problems recur across many projects, such as:

- lack of training and/or experience
- failure to employ a risk engineering approach
- improperly specified contract requirements for software-related items and processes
- inadequate estimates of computational resources needed
- immaturity of Ada development tools and environments
- insufficient incremental testing discipline

While software management methods such as design walkthroughs and code inspections still apply, the levels of effort and time associated with the various phases of the software life cycle change somewhat with the introduction of Ada. Experience to date indicates the need to spend more time in the definition and design phases and less time in testing and integration than was spent in previous efforts. This is probably because Ada enforces strict adherence to certain software engineering practices, resulting in more up-front analysis and design. This is offset by fewer programming changes prior to customer acceptance and easier maintenance throughout the life of the program. Some Ada efforts, such as the air traffic control efforts for the FAA, have experienced increased effort and resources during the design

[26]Life-cycle models include: waterfall model [196]; spiral model [35]; software-first life cycle [128]; risk-driven process model [161]; incremental (or evolutionary) development [1, 69, 42, 164]; and the Ada Process Model [195].

phase (50% to CDR) [175]. Increased emphasis on the design phase, caused in part by increased emphasis on software engineering, will lessen the time involved in testing and integration, since many errors will be identified early in the development [24, 29], and may also decrease the time required in the coding phases [175, 193].

Some rules of thumb to use regarding levels of effort are:

Phase	**Other Languages**	**Ada**
Design	40%	50% +
Code	20%	15%-30%
Test and Integration	40%	20%-35%

In understanding these rules of thumb, it is important to note that NASA's experience [167] indicates that although the major milestones indicating *phases* (such as CDR) changed, the distribution of effort by *activity* (design, code, test) across the phases was quite similar in both Ada and FORTRAN efforts. This shows that software development tasks or *activities* changed little using Ada, but the processes for controlling and managing those tasks (as indicated by *phases* of the development life cycle) were modified as a result of using Ada.

Ada's impact during development has been perceived by many to cause more up-front design effort. This is not so much an impact of the language itself, as it is the impact of more software engineering efforts being applied early in the life cycle. For example, projects have shifted effort into design from later stages, such as testing [29]. They spent more time defining requirements, writing test documentation and user manuals early, but found savings in test and integration. From studies of implementing a spacecraft flight dynamics simulator in FORTRAN, and in parallel in Ada, NASA discovered that a higher percentage of errors was discovered in the Ada project during the implementation phase than in the FORTRAN project, i.e., more errors were found earlier (when cost to correct is lower) [113] and the Ada code had somewhat fewer interface errors [167].

Other lessons learned from Ada experiences are that more computer resources are needed, both host and target, and that configuration management with Ada is more technically demanding, because of the extra complexity added by integrating Ada program libraries with existing configuration management practices.

Cost and schedule are driven more strongly by product characteristics and developer practices than by development languages, although different languages may have differing support environments and productivity characteristics. Factors that have been identified as having an impact on the cost of an Ada project [135, 142] include:

- **Project factors:**
  - system architecture
  - complexity of organizational interfaces
  - required development schedule
  - resource availability
  - security requirements

- **Process factors:**
  - allocation of effort/costs to life-cycle phases
  - degree of standardization
  - scope of support
  - use of modern software methods/practices
  - use of peer reviews
  - use of software tools/environments
  - stability of software tools/environment
- **Product factors:**
  - product complexity
  - requirements volatility
  - degree of optimization
  - degree of real-time requirements
  - degree of Ada usage (percentage of code written in Ada)
  - degree of reuse (percentage of Ada code to be packaged for reuse internally and externally to the project, usage of reusable software)
  - database size
- **Personnel factors:**
  - analyst capabilities
  - applications experience
  - Ada environment experience
  - Ada language experience
  - Ada methodology experience
  - team capabilities

Cost drivers that have a significant impact on Ada projects [142] are:

- degree of standardization
- use of modern software methods/practices
- use of software tools/environments
- stability of software tools/environment

Another key factor is the learning curve, reflected in the experience levels of the project team [142]. Although learning curve effects are typically overcome after two to five Ada projects [193], they must be considered in the planning of an organization's adoption of Ada. Such factors are discussed in the next section.

4.1.2. People

An organization's people are essential in implementing its software process. Enhancement of their skills should be considered in developing management strategies for adopting Ada.

Many acquisition programs using new technologies for the first time are technology transition programs (whether they were planned that way or not), and the acquirers should expect to learn along with the developers. For *both* contractor and customer, implementing a new technology like Ada will require that:

- time be allocated for education and training, and
- technology experts be available for consultation and review.

Very often, it is the program office staff or the program manager on the contractor side who is faced with using new technology. The good manager will recognize this situation and plan for it in several ways, as the use of Ada will bring with it a need for training (both methodology and language).[27] For a successful team, skills must encompass three critical areas:

1. Software engineering skills, including systems analysis, requirements analysis, and design methodology.
2. Ada-specific skills, including Ada design methodology and Ada implementation knowledge.
3. Tools and techniques necessary for implementing specific design methodologies [181].

Management may also need training in the selected Ada design methodology [168]. Existing expectations about development efforts and applying standards, especially those that are rooted in an organization's historical use of another language or another methodology, may be incompatible with Ada technology [168].

4.1.3. Tools and Technology

There are technical issues associated with the adoption of Ada and Ada tools as with the adoption of almost any technology. Software development tools and their use should reach a level of maturity before being used extensively in production.

Tools are valuable aids for increasing productivity and improving software quality. Software tools, including compilers and CASE tools, have a development life cycle very similar to the life cycle of other software:

- requirements development
- design
- product development
- in-house (or alpha) testing
- beta testing[28]
- initial delivery
- continued refinement
- production quality
- maturity
- obsolescence

The maturation process from initial delivery to production quality is a key period and can take significant time. Immature tools (just out of development) can have a negative impact on productivity if software developers must concurrently debug new applications software, new tools, and new hardware. Often new tools are abandoned because they are not understood or they fall short of expectations. Program managers should allow time in program schedules for learning how to use a new tool, developing proficiency, and discovering and correcting problems and limitations.

Some of Ada's early implementations were not sufficiently efficient for specific applications. However, this has changed with the maturation of compilers and tools; today, many large systems would be worse off without Ada. The Ada language itself has facilities for handling complexity; however, one must trade off front-end costs versus life-cycle costs, such as host machine resources and necessary tools and compilers.

[27]See Section 5 for more on the subject of learning Ada.

[28]Beta testing is a limited, constrained test/evaluation performed very close to final completion of a product for the purposes of getting user reaction and finding additional bugs. It is usually performed outside the development group.

Ada *facilitates* (rather than enforces) the application of software engineering discipline. Ada has facilities that allow explicit use of modern software engineering techniques, including modularity, information hiding, data abstraction, and object orientation (to a limited point). It is still possible to write poor code, and Ada can be (and has been) used as a scapegoat for poorly developed and poorly managed system development efforts. However, to gain many of the benefits that Ada facilitates, not only do they have to be used correctly by the practitioners who build the software, but the computer resources on the host computer (memory, processor cycles) and development tools may be more costly overall than typically needed for other programming languages. Ada compilers do more work, in checking interfaces or in separate compilation, for example, than do many other language compilers, and thus require more resources. The belief is that overall life-cycle costs will be lower; this up-front investment will save much more than it costs in the long run. As there is not yet sufficient experience in maintaining numerous Ada systems, this is largely conjecture; data over a 3 - 10 year period are needed before conjecture about maintaining large systems can be confirmed (or denied!).

Throughout procurement (and operational upgrades), it is important to assess the applicability of Ada to a system. If a subsystem is to be upgraded and the interfaces are not clean, nor can they be made clean in a cost-effective manner, maybe Ada is not the right choice. And, if the system is identified as resource-constrained, an assessment of the system to identify those pieces that really are resource-critical can assist in isolating them neatly, and provide risk identification of those pieces where additional managerial and technical attention is applicable.

The goals of a program manager are not to make use of Ada (or any new technology), but are larger: to produce the specified system on time, within budget, usable and maintainable over the projected life cycle, and within the projected scope of the system.

4.2. Adopting New Technology

4.2.1. A Model of Technology Development and Insertion

Any new technology goes through a readily identifiable process of development and maturation. The Ada language and its supporting technology are following this same process. Few adoption issues are unique to Ada. Many are simply issues associated with using any new technology.

Many of the issues that are raised regarding the adoption of Ada are generic concerns encountered when introducing any new technology [27]. Moreover, most issues relating to programming languages can be generalized to more than just Ada.

The process of technology development in both hardware and software generally includes the following phases:

- identification of problem or need
- development and insertion of initial product solution
- iteration:
  - use
  - feedback
  - refinement
- maturity

Ada is generally following this model and is now in the latter phases of the product use, feedback, and refinement iteration; the approval of the Ada 9X standard will signal a major milestone in refinement efforts.

Adoption of new hardware technology has often been faster and smoother than adoption of software innovations. This may be because hardware is tangible, while software is abstract and intangible. Perhaps because of this, introduction of software technologies, including Ada, suffer from a very high degree of inertia. Since introduction of new technology requires change, and human beings are, by nature, conservative, it almost always takes a certain minimal amount of time before new technology can actually be introduced into the workplace.

The use of Ada and supporting methodologies, once demonstrated on a "real" project, can not only serve as a catalyst for the introduction of Ada, but also for other new methods, tools, or techniques [119]. Ada has been called an "enabling technology" — a technology that facilitates the use of other technologies. For example, Ada *enables* building very portable software. Ada also *enables* writing very low-level, machine-dependent software. Both capabilities can be necessary. If bits have to be packed into a hardware-defined message packet and the operating system used to transmit the packet over a specified standard bus, then non-portable code may be needed, and Ada will allow that. At the same time, Ada supports writing code that is very portable.

In understanding software technology adoption, two key points stand out [45]:

1. The transfer of a technology to an organization without the maturity to understand, absorb, and apply it is likely to cause problems, rather than to improve that organization's capabilities.[29]
2. The process of software technology transfer is complex, and it involves many roles and phases.

A generic model of technology transition, such as that in Figure 4-2, shows that technology adoption is a phased activity that moves an *organization* through a sequence of transition stages for a given *technology*. When attempting to introduce a new technology, target audiences within that organization pass through different levels of *commitment* to the technology [158]. The early stages (*contact*, *awareness*, and *understanding*) focus on the acquisition and comprehension of information about the technology; the later stages (*trial use* and *adoption*) focus on relative commitment to actual use of the technology, leading to *institutionalization* or widespread use of the technology throughout the organization. Transition is successful when the target audience reaches the stage of commitment appropriate for that technology in their organization.

When adopting a new technology, more than just the technological factors must be taken into account. Cultural resistance and difficulties must also be overcome. As noted in Figure 4-2, moving through the stages of adoption requires increasing levels of commitment throughout the organization, not just from individuals. The cultural barriers can be pathological; people do not like to change. So transition to any new technology, including Ada, must also address the human issues, the organizational issues, and the process issues.

[29]See [117] for an interesting analysis of how this has happened with Ada and the U.S. Air Force.

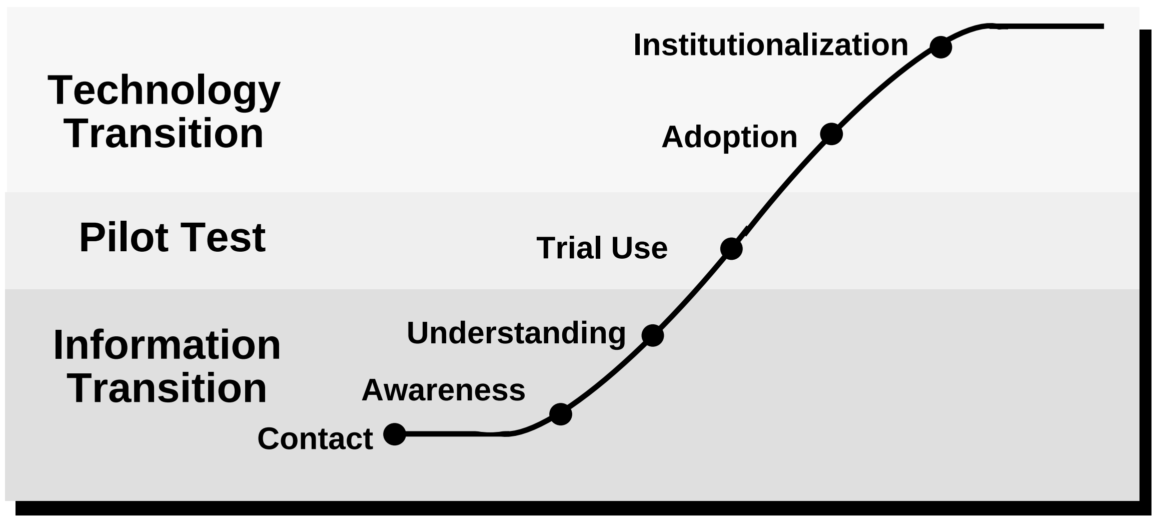

Figure 4-2: Stages of Adoption of New Technology

How does this affect managers? Certainly, a program manager may witness some resistance in dealing with developers; more than likely, their own organization will go through many of the same changes itself. Ada (a "product technology") often brings along with it other "process technologies," such as more modern software engineering discipline or tools to support that discipline. Process technologies invariably affect organizational ways of doing business, so plan on changes occurring.

Ada adoption across the community is at the institutionalization stage in Figure 4-2; however, individuals and organizations may just be starting to move up this curve and address the issues of Ada use.

4.2.2. Inhibitors to the Widespread Use of Ada

As use of Ada has progressed through this process of development and maturation, numerous inhibitors have slowed the widespread use of Ada. These inhibitors can be overcome and planned for in adopting Ada.

Inhibitors to innovation and adoption of new technologies stem from both a lack of awareness and an inability to assimilate the new technology into an organization and its capabilities [45]. Redwine et al. [190] discuss some of the inhibitors to software technology transition. Examples of these encountered in the adoption of Ada are described in [117].

Inhibitors to Ada adoption address many management and technical issues, as well as many myths that developed as a result of early Ada efforts. Many of the inhibitors are not real but only perceived and can be overcome with proper planning and management. The following inhibitors are encountered specifically in using Ada.

Compiler availability: Although Ada compilers are not available for every computer, most major processors in use today, ranging from specialized DSP processors to microprocessors to mainframe computers, have Ada compilers. There are 501 *total* validated Ada compilers listed on the official AJPO list (as of October 1992). This number has grown from 78 validated compilers in May 1987, and only 14 in early 1986. This listing of validated compilers is widely available through online and printed sources (see Appendices A.2 and B.3).

Ada and embedded systems: Although Ada's origins were strongly influenced by the needs of the defense mission-critical computing community, the Ada language design team placed emphasis on supporting modern software engineering practices; the result is a language with wide applicability in developing well-engineered, quality software. In fact, Ada has been used successfully for MIS/CIM applications [87]. There are no technical reasons why Ada cannot be used successfully, and cost-effectively, for such applications [64].

DoD policy and Ada: Many managers of DoD software efforts were not aware of the DoD policy on the use of Ada or believed they were an exception to the Ada policy. Current DoD policy requires that Ada be used for new defense systems and for major software upgrades of existing systems, where cost effective. For efforts that cannot comply with the policy, a waiver must be obtained. See Section 3.2 for a brief description of the waiver process for efforts that cannot comply with the policy.

New technology: Development personnel are cost- and schedule-driven, and they are evaluated on these factors. Using any new technology introduces risks, but those risks can be managed. Now that Ada has matured, the risks to cost and schedule from adopting Ada have been significantly reduced. Recent studies have shown that Ada can be at least as cost-effective as, or more so than, languages that have traditionally been used for developing large, software-intensive systems [39, 193, 81].

Lack of knowledge: A lack of knowledge in software engineering and Ada perpetuates inertia and inappropriate business practices, delaying the transition to Ada. An effective training program is a key part of developing an organization's software engineering capability, and is an area where many organizations find a need for improvement [147]. Ada training, supported by appropriate software engineering training, is available and can assist organizations in improving capabilities. Even though Ada language training addresses a small portion of the knowledge need, software engineering training (in standards, architectures, environments, processes, methods, and reuse) can greatly contribute to the improvement of software engineering practices. This fact is often missed due to a lack of understanding of the relationships between software engineering and Ada.

For example, although software development standards permit tailoring, the lack of knowledge to properly define such tailoring can be an impediment to the use of tailoring; often the perceived incompatibilities between the standards and usage with Ada is blamed on Ada. It is essential, therefore, that there be carefully planned education and training programs that teach fundamental software engineering concepts, design methodologies, and the effective use of the Ada language. Such training programs can be tailored to ensure that system acquirers, as well as software managers and technical staffs, develop an appropriate working knowledge of Ada.

A recent study of software technology transition [32] states that the Ada champion comes from the technical staff 42% of the time, from middle management 36% of the time, and from top management only

14% of the time. It also indicated that a middle management/technical staff consensus was achieved in only about 7% of the cases for Ada adoption compared to 26% of the cases for a more established software technology, such as structured programming. Another recent study [232] has indicated for one large organization that over half of the people in software positions have had *no* Ada education or training. Another study [205] concluded that the low acceptance rate of Ada within their organization is due largely to the lack of a formal education and training program affecting all categories of software professionals. These factors indicate that a successful Ada education program must address all levels in an organization.

DoD procurement process: The current procurement process may not be conducive to Ada adoption and long-term software engineering improvement. A recent survey of Ada adoption indicates that lowest development cost still is the major award factor on DoD contracts, and that industry perceives the DoD as unwilling to trade lower life-cycle cost for greater development cost [49]. This short-term disincentive to educate, improve the process, design for reuse, and engineer for quality is at the root of a range of inappropriate business practices that hamper the evolution of software development toward a true engineering discipline.[30]

The excessive concern for lowest development costs often results in an antagonistic relationship between the contractor and the customer and serves as a disincentive to investment in start-up costs such as training, equipment, tools, and methods. It can also result in unrealistic assumptions (particularly at proposal time) about cost and schedules on an organization's early Ada projects. It may be necessary for the program managers to make creative use of current procurement practices, such as:

- Including Ada and software engineering capabilities, transition activities, or past performance (including quality and reuse) on Ada projects in the proposal evaluation criteria.[31]
- Accounting for increased development costs on an organization's early Ada projects.

Among defense contractors, Ada capabilities are developed in earnest, and Ada is used earlier in producing real systems when organizations believe that they will be more likely to get a government contract [32]. Industry must be convinced that the government is serious about successfully adopting Ada.

Early perceptions: In the face of early disparaging reports on Ada, there has been little advertising of the successfully fielded Ada systems (e.g., [87, 108, 97, 38, 94]) and little concerted effort to gather, analyze, and distribute objective data about the economic impact of Ada on the software engineering discipline. The early bad press has left a legacy because of the weaknesses of early implementations and the experiences on early Ada projects. It is also tempting for all concerned to blame the failure of a project on a new technology. Experiences from successful Ada projects are often withheld by organizations for reasons of competitive advantage [166].

Language issues: Real and perceived language limitations hampered the adoption of Ada. The AJPO has emphasized a strict validation process that has yielded hundreds of validated compilers. Great progress has been made in Ada compiler technology, including the development of optimizing compilers

[30]For further discussion of these issues, see [163].

[31]See [80, 47, 82, 40, 62] for more information about evaluating contractor capabilities.

for many platforms. Clearly, the image of Ada implementations having poor performance and quality is much outdated.

However, many projects still lack the depth of expertise and the understanding of the complex interactions between runtime system (see Section 6.1), language features, and design style to evaluate and select compilers [226] and tools. The quality of compilation systems is an issue of compiler evaluation; evaluation (see Section 6.2.2), as well as validation (see Section 6.2.1), should be emphasized.

The difficulties of interfacing with other languages (e.g., limitations on *pragma*[32] INTERFACE) hinder the introduction of Ada where a large body of reusable software in other languages is available. Some excellent implementation-dependent variations exist but when used may limit future portability.

Previous complaints about the size and complexity of Ada have somewhat subsided to be replaced by laments about Ada's lack of inherent support for OOD, distributed applications, and formal reasoning. C++ and the OSI model are often seen as solutions to OOD and distributed support, although Ada 9X will address many of these concerns.[33]

Tasking is still a concern for some, but continuing runtime performance improvements, Ada 9X enhancements (see Section 9), and rate monotonic analysis (RMA) [204] address many of these concerns. Projects should evaluate Ada implementations in light of their specific requirements.

Integration with CASE tools: Current CASE tools are poorly integrated with most Ada implementations. An integrated development environment should help enforce some of the software engineering discipline encouraged by Ada. Development of such environments is continuing [186]. For example, a standard binding for CASE tools interfacing with Ada libraries (Ada Semantic Interface Specification — ASIS) has been developed and is planned for standardization as an interface for Ada 9X.

4.2.3. Adopting Ada

The real or perceived inhibitors to the adoption of Ada can be overcome by systematically planning and managing the process of learning and using the new technology. Certain considerations can facilitate transition and mitigate risks in transitioning to Ada.

The software industry has been engaged in Ada transition since the early 1980s. Many of the uncertainties faced by the early adopters are better understood today, including:

- Validated Ada compilers are generally available and their quality, though already acceptable, continues to improve.
- Though tools exist, project-selected tools may not be integrated into a software support environment.
- For certain embedded computer systems, real-time designers require control and favor deterministic schedulers not available with Ada tasking [98].
- Research in using Ada for distributed systems is promising but not yet demonstrated for some forms of distributed systems.

---

[32]Pragmas are used to convey information to the compiler, for example, as an instruction to the compiler to perform some special action such as compiler optimization or to interface to software written in other languages.

[33]See [81] for one analysis of the applicability of C++.

- To achieve portability, projects must state portability as a design objective.
- Training for Ada programmers is available; training and experience for Ada designers and managers is crucial.

Managing the transition to Ada adoption is an exercise in risk management, that is, managing uncertainty. There may be uncertainty about the readiness of an organization to accept a commitment to produce software products using Ada. In addition, there may be uncertainty about the specific project needs and their interaction with Ada's capabilities. A management strategy for adopting Ada must incorporate risk management strategies to match an organization's readiness for Ada with specific project needs to identify and resolve any uncertainties.

Thus, if transitioning to Ada is simply an exercise in risk management, why is it difficult? Expertise has been identified as playing a large role in the adoption decisions made by firms with respect to Ada [208]. That study found that firms that have developed greater software engineering expertise appear to value Ada more, and expect to incur fewer costs of adoption. This expertise or sensitivity to software engineering principles, manifested in both contractor and acquisition organizations, was found to be a critical factor in the adoption of Ada. Developing that expertise, with Ada and in the context of an organization's current capabilities, is crucial in an organization's adoption of Ada.

An organization planning to improve its capabilities by adopting Ada must examine its current capabilities and develop risk management strategies for dealing with Ada's impacts in each of the areas shown in Figure 4-1: tasks, people, and tools.

- **Tasks:** Management tasks must focus on planning and controlling costs and schedules, configuration management, product assurance, and measurement; and on the degree to which project plans and estimation factors for product size, productivity, and quality are Ada-sensitive.

  Methodological concerns deal with the degree to which Ada design methods are exploited and how early in the life cycle, how many software engineers are involved, the size of the system to be developed, and the degree to which the requirements are well defined and understood. Risk assessment in this area is influenced by the technological expertise of the organization, the selected life-cycle model, and whether systematic approaches are used for requirements, design, code, and test.

- **People:** Assessment of personnel capability is determined by the degree to which qualified Ada-trained personnel exist. Training needs are determined by the number of software engineers and managers to be trained and the expected knowledge foundation on which that training is based. Existing capabilities and training needs will influence the training resource selected, the extent of the training, and the number of software engineers and managers to be trained. Organizations that have used in-house training have developed their Ada capabilities faster [32].

- **Tools:** Selection of tools can be very important in adopting Ada. The selected and acquired Ada compiler must meet the host and target needs of the project in terms of the capacity requirements (time and space) for both host and target systems, the performance demanded by the application domain, the computer system architecture selected for the target system, and requirements for growth and adaptation. An active process of compiler specification, selection, and acquisition can aid in this evaluation and selection (see Section 6.2.2, as well as [226]).

  The extent to which Ada and CASE tools support the technical and management activities of the life cycle and the degree to which these tools operate as an integrated environment during both product creation and post-deployment support is another important factor. Each activity of the life cycle must be assessed for tool and methodology congruence.

The organization that wants to adopt Ada as a programming language and recognizes that Ada adoption must be managed may find these considerations useful in planning and managing the transition to Ada. The question no longer is, "Is Ada ready?"; but rather, "How ready are we for Ada?" Organizations that acquire Ada capabilities and an Ada compiler early in this adoption effort are using Ada sooner for production work [32].

4.3. Managing Ada Adoption Efforts

An organization successfully adopting Ada will invest in, plan, and manage Ada adoption efforts as it moves through the various stages of adoption.

Introducing Ada effectively necessitates that an up-front investment be made to reap longer term benefits. There are several classes of economic investments and cost drivers:

- Compiler and other tools.
- Hardware testbeds.
- Development hardware costs, as routine computer facility expenses can be somewhat higher than expected because of increased compile time and storage costs.
- Software development environment costs.
- Training costs, including language training, and training in new software engineering practices, disciplines, and methodologies for management and software personnel in developer and acquirer organizations.

Several of these costs are one time in nature and will decrease as Ada training is completed, Ada programming environments are widely installed, as software tools mature, and as hardware cost/performance continues to drop. Although incurring up-front costs is acceptable, it presents a special problem for Ada acceptance by program managers and contractors. The costs of training, compiler and associated tool acquisition, and associated hardware are real and readily measured, while the benefit is more difficult to measure and will accrue in the future over the life cycle of the software and subsequent projects. Program budget and schedule must reflect the costs and risks.

Program managers need to understand several key points:

- Education for software engineering and software project management is *required* for *both* customer and contractor.
- Software discipline and planning must be enforced. For example, the use of a common program formatting style is a simple technique to improve the readability, understandability, and maintainability of the code. In addition, a common style can be enforced using a tool known as a pretty printer (see Section 6.2.2.3).
- Periodic risk assessments should be planned and executed.
- Performance issues and risks should be addressed (see Section 7.4.1).

Organizations just developing Ada capabilities will often develop their initial capabilities through training and the acquisition of compilers and tools that meet project needs. Programmers will be trained to design, read, and write Ada programs, and managers will be trained to read Ada designs recorded in the style of the project, as well as in management topics. Perhaps most important is the development of an internal core group of skilled Ada personnel.

The following action plan is recommended for sprinkling a larger effort with personnel drawn from this initial core group of skilled personnel:

- Develop and execute a long-term, phased training plan that includes management, application specialists, lead designers, software engineers, and testing personnel. Management training is a must. Course material must include software engineering principles and application concepts. Insights into various training strategies can be obtained from Section 5 and [181].
- Structure the development schedule to contain a few non-critical Ada tasks that can be accomplished very early in the program (in parallel with requirements definition and preliminary design). The main purpose of these tasks, which could be prototypes of parts of the system under development, is to develop some personnel experienced in using Ada. Emphasis should be on:
  - small team efforts,
  - extensive review of the ongoing development,
  - document lessons learned and distribution of this information, and
  - management observation.
- Since Ada has been shown to change the levels of effort applicable to various phases of the life cycle, management should not attempt to rigidly control the development of these tasks, but rather should use them as a vehicle to understand what will happen in the mainline development.
- Sprinkle the experienced personnel from the small Ada projects in as leaders of the mainline development [191].
- Based on the results of the early Ada tasks, adjust the software development plan (SDP) and the training plan for the mainline effort.
- Consider bringing in some outside experts with experience developing similar systems in Ada to evaluate the system design (hardware and software) and implementation strategy.

An organization that is institutionalizing its Ada capabilities is able to make effective use of its management processes and methodology mechanisms to develop Ada software. The organization has selected a life-cycle model, one that exploits Ada in the earlier activities of the project. For each activity in the selected life cycle, appropriate methods and tools have been chosen and personnel trained in their use. Organizations that have accomplished this will often look to other advantages to be gained through reuse or domain architectures.

Gaining the benefits of software engineering does not happen magically. Both the customer and the contractor must be educated in software engineering and its management; both must be actively involved — by setting a good example — in the application of sound software engineering and management practices. And even if the management is excellent and the requirements are stable, risk assessments (of Ada, of system requirements, and of new, upcoming technological solutions) should still be done.

4.4. For More Information . . .

The resource recommendations below are intended as an initial starting point for those looking for more information about the topics addressed in this chapter.

Software Process Management

- *Capability Maturity Model for Software* [183]
- *Key Practices of the Capability Maturity Model* [222]
- *Software Capability Evaluation (SCE) Tutorial* [47]
- *Managing the Software Process* [125]

Software Process Improvement

- *Software Process Improvement at Hughes Aircraft* [127]
- *Software Engineering Process Group Guide* [110]
- *Implementing Software Engineering Practices* [44]
- *A Guide for Implementing Total Quality Management* [66]
- *An Introduction to the Continuous Improvement Process: Principles and Practices* [159]
- *Implementing Total Quality Management: An Overview* [137]

Ada Adoption

- *Software Engineering as a Radical Novelty: The Air Force Ada Experience* [117]
- *Process Maturity as a Guide to Phased Ada Adoption* [172]
- *Ada and Software Management in NASA: Assessment and Recommendations* [8]
- *Adoption of Software Engineering Innovations in Organizations* [32]
- *Understanding the Adoption of Ada* [208]
- *Software Technology Transition* [158]
- *Ada Risk Handbook* [212]

Ada Project Management

- *Department of the Navy Ada Implementation Guide* [176]
- *Experiences in Delivering a Large Ada Project* [50]
- *TRW's Ada Process Model for Incremental Development of Large Software Systems* [195]
- *The Second Ada Project: Reaping the Benefits* [152]
- *What Every Good Manager Should Know About Ada* [27]
- *Advanced Automation Systems Experiences with Ada* [101]

5. Learning Ada: Training Implications

5.1. Rationale for Ada Training

Maximizing the benefits of Ada requires knowing the software engineering techniques and disciplines that Ada supports as well as the language. It is more expensive to teach software engineering and Ada than to teach just Ada, and it is necessary to teach software engineering in conjunction with the introduction of Ada. Of course, Ada training will vary among employees as not everyone needs the same level of expertise in Ada and software engineering.

An effective training program is a key part of developing an organization's software engineering capability and an area in which many organizations find a need for improvement [147]. A training program involves identifying the training needs of the organization, the projects, and the individuals, and developing and procuring training courses to address these needs [183].

Appropriate software engineering training, supported by appropriate levels of implementation training (Ada, other languages, tools, etc.), is required to realize Ada's positive impact on schedule, cost, and quality goals. An effective training program should be based on the following points:

- Using Ada effectively requires more than just knowing the language — it requires understanding the software engineering techniques that the language supports [217]. For example, data abstraction and information hiding, while easy to understand in principle, may present some difficulty in application [140]. As a result, quality Ada training must be different from and more extensive than training for other languages. While Ada training will cost more initially because more will be taught, in the long run, the use of Ada and its associated software engineering concepts can reduce individual program development and maintenance costs.
- Training is necessary for *both* acquisition and development/maintenance personnel. Software designers, system engineers, Ada programmers, and software managers in both industry and government *all* need to know about Ada and its associated software engineering techniques. Ada training should address all these groups, but the depth and kind of knowledge needed will vary for each. Because of Ada's new concepts, the learning curve for all concerned can be steep.
- Management training is especially critical. To capably manage and direct programs involving new technology, program managers must understand how to apply the technology [205, 23, 140] and how it differs from existing approaches. In particular, managers should understand how to acquire and apply metrics in software development [30, 197, 48]. Ada projects have different profiles from projects in other languages and managers should be aware of this in tracking and evaluating metrics.
- Selecting or creating the training program for all levels deserves special attention. Many trainers have excellent, well-integrated courses available; many teach only old programming methodologies and concepts wrapped up in Ada syntax. Teaching Ada correctly, from curriculum development through course delivery and evaluation, is a difficult and time-consuming task. The right trainer and training course can be hard to find.
- It is useful for a project to build a core group of knowledgeable, well-trained individuals to serve as mentors for more junior personnel. These individuals must be very knowledgeable in both the application of Ada and software engineering, and in the project domain. Experience has shown that a few such individuals, placed in key positions within the project, can have a stimulating effect on other engineers.

In the absence of any recognized standard for evaluating the abilities of software engineers, training will continue to be the responsibility of the project manager. Managers may have to select personnel using project-specific criteria and then train them to achieve the required level of software development expertise.

5.2. Audiences and Course Content

All personnel need some training in Ada and its associated software engineering concepts to obtain the maximum benefits from using Ada. In addition, an Ada training program should always consider the background of students and the goals of the project.

There are several target audiences, each with their own perspectives, that may require education or training [205, 192]. These include:

- **Management personnel**
  - **Executives:** focused on business issues
  - **Software managers:** focused on technical issues and risks
  - **Non-software managers:** focused on management implications of Ada use
- **Technical personnel**
  - **Engineers:** including systems engineers and top-level designers
  - **Programmers and analysts:** including implementers and testers
  - **Project support personnel:** including administrative, configuration management, and software quality assurance personnel
  - **Trainers**

Following are some guidelines regarding target audiences and key content material:

- **Prior programming language:** The background of students attending the classes must be considered in course design. For example, software engineers who have not used languages with dynamic data capabilities (i.e., access types) will probably need more extensive training than those with prior exposure to these concepts. Examples of languages without these capabilities are assembler and FORTRAN. Such individuals can be easily overwhelmed in training courses that are too intensive and too short. Initial training for these individuals can take 3 weeks; becoming truly productive using Ada can take 4 to 6 months of experience. Those who already know a structured high-order language like PASCAL should assimilate the new techniques more rapidly. However, beware of teaching by analogy (i.e., by showing a construct in an old language and then demonstrating how to accomplish the same thing in the same way in Ada). This leads to a continuation of the problems of earlier software development generations written in new syntax, e.g., AdaBOL or AdaTRAN. The student is less likely to readily adopt new ways of accomplishing old tasks that are more powerful and more suited to Ada, taking advantage of "new" Ada features such as packages, tasking, exceptions, aggregate assignments, use of Boolean expressions, etc. Students must be taught to think in Ada [202].
- **Prior training:** The knowledge and experience of students should be considered; prior attendance in programming courses, software engineering courses, or software acquisition courses cannot be assumed to have prepared students adequately for project needs. Certain aspects of software issues may have been taught, but the inclusion of appropriate Ada material should not be assumed. Indeed, a paradigm shift is required to think in terms of libraries and reuse as first options.
- **Software engineering:** All personnel need an introduction to the concepts of software engineering [217]. System and software engineers and software testing personnel need an in-depth exposure to the concepts, principles, and practices of software engineering. In addition,

these development personnel will need to be learn techniques for implementing these software engineering concepts in Ada.

- **Ada programming language:** Although the management and technical audiences described above need Ada training, the content and level of the training may vary significantly with the audience [192]. Managers need an overview and an understanding of Ada capabilities, risks, and impact on their processes, while software engineers, systems engineers, and software test personnel will need more extensive Ada training.
- **Software design:** Engineers concerned with software design will need training in the use of Ada-based program design languages (PDL). In addition, there are several new methodologies that take advantage of the concepts in Ada as part of the design process [211, 58, 53, 54, 65, 55]. The challenge for any training program is to provide effective training for software engineers to design systems that are elegant, while stressing the implications of the design options to performance concerns. Designers need to become especially cognizant of the long-term maintenance costs of their design efforts.
- **Compilers and software tools:** Software engineers developing Ada software for embedded systems need to understand how the Ada compiler, runtime system, and tools used on a project affect memory and throughput constraints. In particular, in embedded systems programming there may be a need to use compiler vendor-supplied routines to integrate with the operating system or runtime environment. These needs should be examined for an Ada solution (see [151] and Section 8.4) before interfacing to another language to obtain this runtime support.
- **System engineers:** System and software engineers responsible for hardware and software integration will need a thorough understanding of the systems engineering impact of high-order language selection. Tradeoffs to be considered include: whether a compiler is available for the chosen processor, its capabilities, and its requirements in terms of space and speed.
- **Software testing:** Software testing personnel need to understand how Ada language constructs such as packages, generics, and tasks affect testing approaches [216]. In addition, separate compilation in Ada affords earlier testing of some modules through drivers.
- **Configuration management:** Configuration management personnel need to understand how Ada language features such as packages and program libraries affect configuration management procedures. They may also require special training on tools that provide configuration management for Ada software systems. The addition of libraries for reuse purposes may also complicate the configuration management aspects of new development projects. These concepts should be part of the training for software engineers as well.
- **Management:** Managers should obtain an understanding of software engineering principles, Ada concepts, the impact of methodologies on the software architecture, the impact of Ada on the software and system life cycle, the costs and capabilities of tools, and hardware and software interaction. Senior managers may not need to know detailed language syntax, but they may need an understanding of certain language concepts, such as the *package* concept (to appreciate Ada's defined support for modularity), or *exceptions* (Ada's mechanism for error handling). This becomes critically important if design reviews or inspections are done using Ada/PDL or Ada code. Managers should be taught the means whereby metrics can be acquired and used in the development of a software project. Understanding these concepts may improve the overall understanding of a development or maintenance effort. In addition, the total quality concepts (i.e., TQM or TQL) must be understood as part of a continuous effort to improve the end product through software process improvement.
- **Design reviews:** Personnel attending design reviews should have some understanding of how Ada will affect the software architecture. Design cannot always be language-independent if the language affords a capability worth exploiting in the design.
- **Subcontractors:** Training requirements should be consistently applied throughout the organization, as well as by any subcontractors. This will help ensure continuity in methodology, standards, and approach across the entire development effort [140].

5.3. Evaluating Training Programs

The most important aspects of any Ada training program are a software engineering emphasis and hands-on practice. The quality of the course material and the background and experience of the instructor are also important considerations.

Following are some guidelines for evaluating Ada training programs:

- **Tailoring:** Training can be purchased or developed internally. Curricula developed internally or customized for an organization tend to be more effective because they:

  - provide for curriculum coordination
  - can embody specific methodologies and standards
  - can be oriented toward specific tools and hardware considerations
  - are generally more responsive to sudden, short-term needs

  The price for this extra effectiveness is continual improvement of the material through the introduction of new material and ideas. Internal trainers will need to be assigned full-time responsibility for preparing and updating this training.

  Training prepared by in-house personnel is industry's preferred way for Ada training and was used by 36 percent (the largest percentage) of the organizations surveyed [32]. Extensive use of training prepared by in-house personnel while acquiring an organization's first Ada compiler or while developing an organization's Ada capability has been shown to be significantly related to early use of Ada for software development projects [32].

- **Emphasis:** The emphasis of courses should be on understanding what the language can do and how it should be used to produce well-engineered systems. Courses that teach *only* the syntax and semantics (the grammar and meaning) of Ada should be avoided. Course content should emphasize software engineering principles with Ada examples showing how to implement them. Care should be taken to evaluate the training program to insure that the entire life cycle and its implications are considered.

  Training courses often tend to focus solely on the coding phase. There are two separate schools of thought on this problem. One is to have the student begin writing code immediately so as to provide some immediate gratification, then the rest of the life cycle will become meaningful. The other school of thought stresses that students need to know the implications of what they are doing *before* they start writing code. In general, the second school of thought is the preferred approach.

- **Length and topics:** For any introductory course, a *minimum* of five class days is required to teach the basics of the language and to introduce the underlying software engineering principles. These class days should be spread over a minimum of two weeks and reinforced with hands-on design and programming exercises. Additional classes, which include Ada's more sophisticated features and implementation tradeoffs, will be required for people who will use Ada heavily, who do not have previous HOL background, or who will be responsible for design activities. Refresher courses should be offered periodically after the initial training.

- **Equipment:** All training should be on the specific compiler and tool set to be used in the development effort. The user interface for different vendor compilers is significantly different, and there is no standardization on the means of managing libraries or invoking tools. Thus, to avoid retraining costs and time, it is advisable to conduct the training on the actual compiler and tool set to be used. Only validated compilers should be used.

- **Documentation:** In addition to handouts of course materials, model solutions to problems should be available; they are an especially useful way to teach the Ada language and its underlying principles. The contents of reuse libraries can also be used for familiarizing students with the tools at their disposal.

- **Hands-on lab:** It is *imperative* that hands-on programming exercises be a major component of the training process. Just as it is difficult to learn how to repair radar sets or jet engines without practical, applied work, the same is true for building computer software and learning computer languages. In fact, the more detailed the training, the more imperative hands-on exercises become. For example, in classes dealing with using Ada for a specific embedded processor, there is no substitute for writing software with the compiler and tool set that will be used on the actual development program. The software engineers working at this level need to understand the performance and limitations of a particular tool set.
- **Monitoring progress:** The training program should have methods for measuring student progress, providing student feedback, and evaluating course effectiveness.[34] Since hands-on training is essential, a careful and complete review of each programming project should be made by the instructor. This will include *detailed* comments on the design approach, the selection of Ada features used to accomplish the design, and the use of new Ada concepts where appropriate.
- **Instructor availability:** A trained instructor must be available to help students and provide guidance when required. The instructor must have a wide-ranging experience in both the use of Ada and in design problems for the likely domain. Finally, the instructor must be well versed in the concepts of software engineering. Practical experience in the development of software-intensive systems would also be particularly valuable for the instructor.
- **Computer-assisted instruction (CAI) and video tapes:** These are good *supplemental* tools for teaching language syntax and providing overviews and review. However, they should *not* be the focus of the training effort because they generally do not have a software engineering emphasis. Even when they do, nothing can replace the effectiveness of good classroom instruction.
- **Real project use:** To achieve maximum training effect, personnel should be assigned immediately to an Ada project where experienced Ada software engineer(s) are available for consultation for the first 4 to 6 months. If the student does not apply the new knowledge soon after instruction, the knowledge rapidly diminishes.

  In addition, some organizations have improved training success by training project teams. These teams can also be exposed to continuing training which stresses applying Ada within the context of the project's processes and infrastructure.

5.4. Selecting Training Programs

An Ada training program should be planned and selected based on an understanding of the learning needs of those who will be trained and the requirements of their projects.

As discussed previously, there are many factors to consider when selecting a training curriculum, not the least of which is the material to be covered, the depth of training, who should be trained, and the qualifications of the instructor. As a general guide, the following items can be used to prepare for the selection of appropriate training:

- A thorough curriculum outline for each different learner population in the organization, with learning objectives, content outline, instructional strategy, and suggested training media (at least two general populations can be anticipated: management and engineering).
- A vendor selection guide that identifies how to select vendor training materials for the Ada training. This is not an easy task and could take several weeks to create. The guide must be

[34]A possible approach here is the use of before and after course testing.

predicated on a thorough needs assessment of the populations and the curricula tailored accordingly. This is a time-consuming and labor-intensive effort, requiring investigation and analysis for each learner group. All persons to be trained will need software engineering concepts[35] as a significant part of any Ada training.

A guide with two parts should be created for this purpose. The first part should discuss general issues relevant to vendor selection and can be prepared to be applicable organization wide. The second part should discuss detailed considerations of how to customize the first part to tailor it for a specific project or group within the organization.

The selection materials can take the form of a document and briefing for senior management in the organization, with a more detailed planning document developed for project approval and use. This planning document should contain a statement of need, appropriate project objectives, a suggested approach to the task, expected benefits, and expected product.

Other factors that an organization may consider in selecting a vendor include:

- Third party references: observations from people from different organizations that have taken the training.
- Evaluations of selected individuals who attend the training before committing an organization to its use.
- Availability of "train the trainer" training.

5.5. For More Information . . .

The following resource recommendations are intended as an initial starting point for those looking for more information about Ada training. Refer to Appendix B for contact information.

- *Ada: Helping Executives Understand the Issues* [220]
- *The Pedagogy and Pragmatics of Teaching Ada as a Software Engineering Tool* [174]
- *Practical Methods for Introducing Software Engineering and Ada into an Actual Project* [41]
- *Holistic Case Study Approach to Ada-Based Software Engineering Training* [181]

Ada Information Clearinghouse (AdaIC): AdaIC has information about upcoming classes, seminars, etc. For more detailed information, the AdaIC publishes the *CREASE: Catalog of Resources for Education in Ada and Software Engineering* [134].

Ada-JOVIAL Newsletter: This newsletter is published at Wright Patterson AFB, Ohio. Several pages in each issue are allocated to training and other product announcements.

Ada Software Engineering Education and Training Team (ASEET): This group examines issues in providing quality and timely Ada education and training. The team is producing a database of information on Ada training and education material known as the ASEET Material Library (AML). The AML consists of course syllabi, lecture notes, overhead transparencies, books, articles, notes, etc., that can be obtained by DoD personnel and copied at cost by non-DoD personnel. All courseware is non-copyrighted.

[35]How much content in software engineering will be needed can be expected to vary with the different learner groups.

6. Software Production Technology

6.1. Software Development Terms and Concepts

The following terms and concepts are explained in this section: host/development computer, target computer, embedded computer, source code, object code, runtime software, program library, compilers, linking, debugging, downloading, and program execution.

This section introduces generic software development terms and concepts discussed in the rest of this handbook. To focus on areas of critical importance and to provide a basis for subsequent discussions, this section concentrates on the process of transforming *source code* developed by a software engineer into *object code* that will *execute/run* on a target computer. The process of code transformation is a small, but vital, part of the overall systems development life cycle. *Readers already familiar with these items can skip to Section 6.2, which discusses Ada compilers.*

An Ada implementation (or an Ada compilation system) is an Ada compiler, linker, library manager, run-time software, and any other necessary software associated with its host computer and the target computer. These two computers are used in the process of software development:

- **Host (or development) computer:** The software development process typically takes place on the host or development computer. This computer, whether it be a large mainframe computer or a personal workstation or a personal computer, serves as the *host* for the development software.
- **Target computer (or target processor):** The computer that runs the developed software; embedded systems examples are various RISC processors, MIL-STD-1750A, Intel 80x86 family, and the Motorola 68000 family.

While the host and target computers can be the same, the host computer is often different from the target computer because:

- More (and different) resources may be needed to design, develop, document, compile, and maintain software than to execute it. Such resources might include system memory and peripherals such as disk drives.
- In many systems, such as weapons systems or other real-time systems (air traffic control), the target computer is *embedded* in and functions as an integral part of the system. Embedded computers are typically small, special-purpose machines dedicated to specific operations or functions and therefore not suitable for developing software. Examples of embedded computers are processors found in automobiles, household appliances, industrial robots, navigation systems, and process control equipment.

When the target computer is different from the host computer, the executable code must be transferred (*downloaded*) to the target computer before it can be run.

As Figure 6-1 shows, the software developer uses an interactive software tool called an *editor* to create and save (and later modify) the source code for each of the many units (or modules) that will constitute the completed system. The source code is, for the purposes of this handbook, written in a high-order language (HOL), a programming language such as Ada, COBOL, JOVIAL, or FORTRAN. HOLs enable a software engineer to write in an English-like, readable form rather than in a binary machine language unique to each type of computer system. Not shown in this figure are various front-end CASE tools that aid in designing the system.

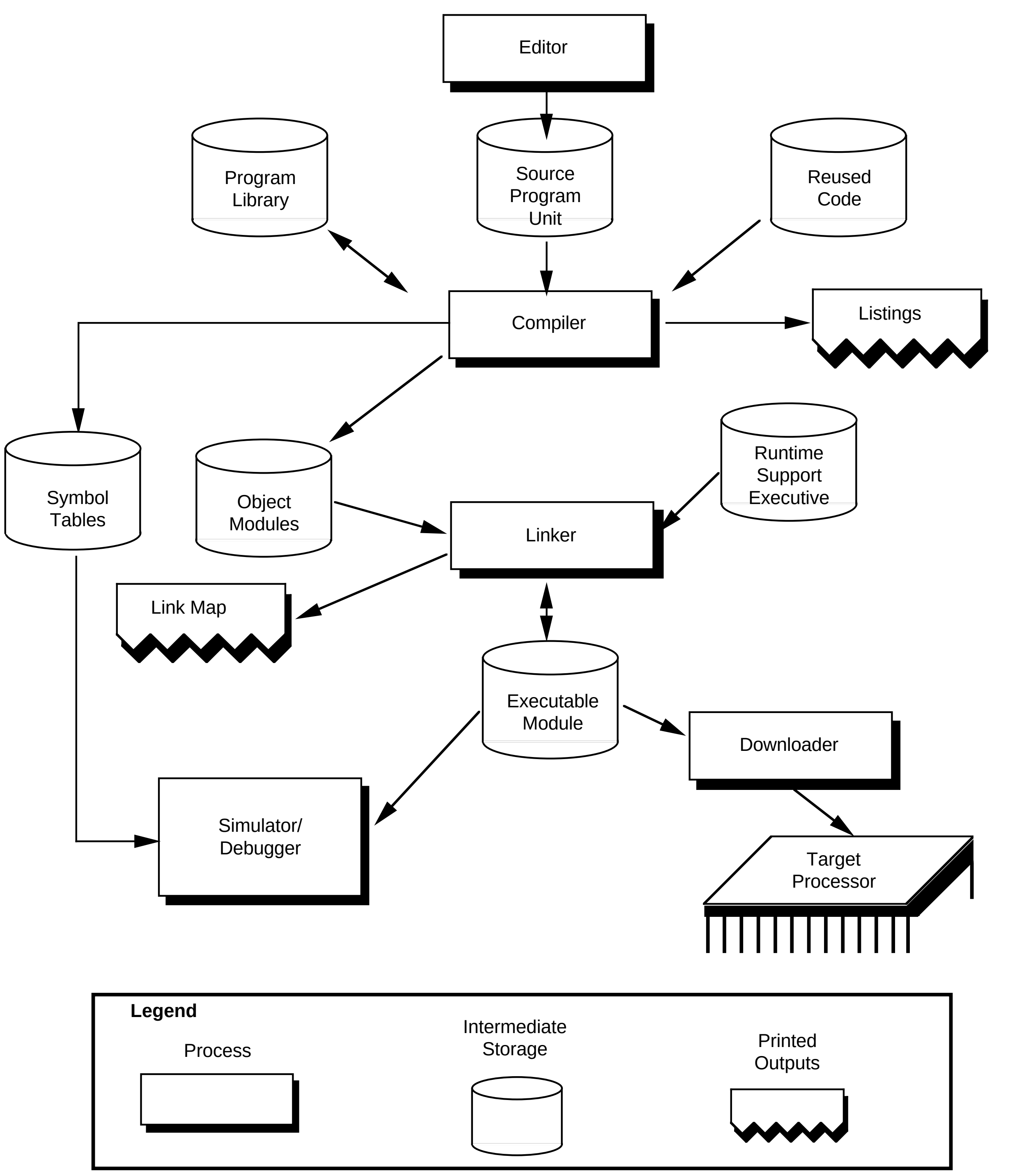

Figure 6-1: Tools Used in the Software Development Process

After each unit is created, a *compiler* checks the source code for first-level correctness by ensuring that the syntax (grammar) of the code is correct. If errors are identified, the process cannot continue, and the developer then uses the editor to correct the error(s) and resubmits the module to the compiler. If no errors are found, the compiler:

- translates the source code into *object modules* that will (eventually) run on the target computer (in applications where often the host and target computers are different, a compiler that translates source code into a machine language for other than the host computer is known as a *cross compiler*),
- produces various output listings (see Section 6.2.2.3), and
- produces symbol tables that are used in conjunction with debuggers.

Relationships among object modules (and in some cases, the object modules themselves) are maintained in a *program library*; libraries vary in their sophistication, depending on the requirements and capabilities of the high-order language being used. Ada libraries are much more sophisticated than libraries for other languages.

The independently compiled object modules must be *linked* together before the program can be executed. When the application code is linked, the appropriate *runtime software* is automatically included. Runtime software provides the additional supporting functions required for executing programs on a specified target computer. Each target computer and each computer language requires its own specific runtime support library. For Ada, functions most likely handled by runtime software include scheduling, parameter passing, storage allocation, and some kinds of error handling. As part of the linking process, additional consistency checks are performed; in particular, any missing or possibly obsolete compilation units are reported. The outputs of the linking step include:

- an *executable* (ready to run) module
- printed listings, generally called a link map, that describe the structure of the executable module

The executable module can be executed and tested on either the host or the target machine. Usually, the host machine will continue to be used, especially during the early phases of software debugging and testing, because of the greater resources available on the host. If the host machine is not the target computer, it is possible to simulate execution of the developed software by using a *simulator/debugger*. The simulator/debugger, which is also referred to as a target simulator, runs on the host machine and mimics the timing and functional behavior of the target machine, thereby providing a good view of execution in the target domain. An additional level of problems can then be discovered *before* the software is downloaded to the target. If logic errors (errors in functionality and operation) are discovered, the software developer must return to the editor to begin the correction process.

After a certain confidence level is reached using the host-based simulator/debugger, the ultimate test is to download the executable software to the target computer and execute it there. While simulators or emulators are intended to mimic the timing and functional behavior of the target machine, they may have some differences in performance or execution that may mask problems. There is no substitute for *hot* testing software (i.e., testing on the target processor). On the target processor, debugging tools are very helpful for locating errors. Errors discovered during software execution are the most expensive to resolve.

6.2. Ada Compilers

Initial Ada compilers primarily focused on adhering to the language standard and/or establishing a market presence. Subsequent generation, production-quality compilers are available for many more processors, and are focused on optimized object code. Some important issues to consider when selecting an Ada compiler are compiler validation and compiler evaluation.

While all tools used in the software development process are important, the Ada compiler continues to attract the most attention, in part because of:

- the requirement for compiler validation to ensure that a compiler conforms to the language standard; and
- the need for compiler evaluation, which should be used to determine the usability of a validated compiler for a particular application's needs.

These issues are briefly addressed below. The *Ada Adoption Handbook: Compiler Evaluation and* Selection [226] addresses these issues in greater depth.

6.2.1. Compiler Validation

To ensure that program-dependent dialects of Ada do not arise, the DoD requires that validated *Ada compilers be used for creating software to be delivered to or maintained by the government. However, not all validated compilers may be suitable for every program. Each program must select an appropriate compiler and tool set, and make allowable improvements when necessary. Compiler modifications are acceptable if they do not change the language. Modified compilers must be revalidated, but at the discretion of the program manager.*

While Ada has been mandated as a standard by the Department of Defense, it is not enough just to have a standard — the standard must be enforced. The purpose of validation is to require conformity of Ada implementations with the language standard. Currently, validation measures conformity with the current language standard [13]; after the adoption of the Ada 9X standard, the validation process will measure conformity with the Ada 9X standard. The official AJPO list (October 1992) contains 270 validated *base* Ada compilers and 231 *derived* compilers,[36] which are available for a wide variety of host-target combinations. In general, only validated Ada compilers can be used for DoD applications.

[36] Once an Ada compiler has a current validation certificate (the base compiler), then variations or modifications to that compiler and/or its configuration result in a derived compiler. A derived compiler must conform to the Ada programming language in every respect and by the same measure as does the base compiler from which it was derived. To be considered a derived compiler, and therefore acquire status as an Ada validated compiler, derived compilers must be registered by vendors with the AJPO. Example: Consider a validated compiler hosted and targeted to a VAX 750. The vendor could then indicate that the same compiler running on VAX 780, 785, 8600, 8800 and other members of the VAX family is derived because it meets the four criteria for derived validations:

1. the validation certificate for the base implementation has an expiration date at least three months beyond the time of derivation;
2. the host and target computer systems of both the base and derived Ada implementations have compatible instruction sets and operating systems;
3. the derived Ada implementation contains an Ada compiler that was obtained (*derived*) from the Ada compiler of the base implementation;
4. the ACVC result for the Ada implementation is either the same as or has minor justifiable differences from the base implementation.

For more details, refer to the *Ada Compiler Validation Procedures* [3].

The validation process is carried out by the Ada certification body consisting of the Ada Joint Program Office (AJPO) for overall direction, the Ada Validation Organization (AVO) and the Ada Compiler Validation Capability (ACVC) Maintenance Organization (AMO) for technical support, and the Ada Validation Facilities (AVF) for performing validations.

There are two ways of obtaining validated status: by AVF testing; and for "derived" compilers only, by registration. With Ada, validation by testing is achieved by testing for compliance with the ACVC, an integrated set of tests, procedures, software tools, and documentation used to validate Ada compilers. [37] After a compiler has been validated by on-site testing at an Ada Validation Facility (AVF), the AVF issues a Validation Summary Report (VSR), which reports the results of the compiler's validation testing.[38] Expiration dates for these validation certificates is twelve months after the expiration of the ACVC version used to obtain the certificate.

Since first introduced in 1983, the ACVC has been extended and revised to increase both the depth and breadth of the test coverage. The first version of the ACVC had approximately 1700 tests; the current version V1.11 has 3719 tests. If a compiler passes all the *applicable* ACVC tests, it is said to be *validated*. Depending on the characteristics of the target computer, some tests may not be applicable. Current policy regarding Ada validation is tending toward a more stable test suite based on expected usage. After removing tests of little value and adding tests for Ada 9X, the suite will stabilize at about 4000 tests.

Although ensuring that Ada implementations conform to the standard has a long-term benefit, the value of validation to an individual program is limited. Validation should be regarded as a necessary, but not sufficient, condition. The program manager should know that:

- Validation does not mean a compiler is free of errors — no tests can detect *all* errors in a complex software product.
- Revalidation is required periodically. The general rule is that a compiler must be revalidated under each new release of the ACVC to retain its validation certificate, but the rules for project compilers incorporated into a project baseline allow this compiler to be used throughout the life cycle of the project.
- The validation process is *not* concerned with performance or usability issues such as compile-time and object-code efficiency (see Section 6.2.2). Validation implies only a minimal level of usability, a level that may not meet the needs of a particular program or application. Consequently, each program must determine whether a validated compiler meets (or can be modified to meet) its needs.

It is not always possible to have the final delivered system compiled with a validated (under the current validation suite) Ada compiler. Validation policies [3] include the concept of a *baselined project compiler* — a particular Ada implementation selected for use for the life of a particular project. It may be necessary to ensure proper and systematic implementation baselining of the software development environment, because that compiler and environment will be used upon system delivery to maintain that system.

---

[37]The ACVC test suite is available through the National Technical Information Service (NTIS) Federal Computer Products Center, U.S. Department of Commerce, Springfield, VA 22161.

[38]Copies of these Validation Summary Reports can be obtained from either the NTIS or the Defense Technical Information Service (DTIC), Cameron Station, Alexandria, VA 22304-6145.

The requirement to use only validated compilers may appear to pose a risk when combined with the DoD requirement to revalidate compilers periodically. After all, if a program's compiler fails revalidation, it must be corrected, and the application code must be recompiled and completely retested, which can have cost and schedule implications. The validation procedures and guidelines document addresses these concerns by providing the following guidance:

- Any compiler placed under baseline project control must be a validated compiler. Once placed in the baseline, the compiler does not have to be revalidated until the program manager finds it prudent to confirm that the compiler still conforms to the standard.
- Program managers are encouraged to use the validation tests to ensure that compiler maintenance changes and modifications have not affected the ability of the compiler to pass the tests.
- Retesting is recommended at major program baseline milestones.

In short, revalidation testing is completely under the control of the program manager and can be scheduled at times when it is least disruptive to the program. The intent of the AJPO guidelines is to ensure that compilers continue to conform to the standard without imposing an undue burden on program managers.

Validation ensures conformance with the ACVC test suite, but is no substitute for evaluation. Evaluation ensures compatibility between the capabilities of an Ada implementation and the project's needs.

6.2.2. Compiler Evaluation

A number of factors are indicative of compiler quality and usability. A project should select a compiler after a thorough evaluation based on its specific needs. In some cases, compiler improvements and/or customization may be needed for certain programs.

The validation process is *not* concerned with performance or usability issues such as compile-time and object-code efficiency. As validation only measures conformance to the language standard, compiler evaluation efforts should determine the usability of a validated compiler for a particular application's needs. Each program must determine whether a validated compiler meets (or can be modified to meet) its needs.

The production quality of an Ada compiler is usually measured minimally in terms of:

- **Compile-time efficiency**: Speed at which the compiler transforms various language features and compiler capacity in terms of the size of components that can be processed and generated.
- **Object code efficiency**: Size and speed of the resulting object code, including runtime software. These factors are often compared against assembly language for the same algorithm.
- **Compiler services**: Quality of error messages, types of error recovery, output listings, and diagnostic information.
- **Support for embedded system requirements**: Ability to precisely control and interface with hardware system functions.

The following sections address these issues. Other factors that should be considered include the number of known bugs or limitations in the compiler (if any), stability of vendors and their products, documentation, and vendor support.

Each program should consider these factors in evaluating the quality and maturity of compilers based on their criteria. The Ada Compiler Evaluation Capability (ACEC) [100, 153, 154, 155] provides criteria and tests for evaluating Ada compilers.[39] Another method of compiler evaluation that has been used is embodied in the Ada Evaluation System (AES), which was developed in the United Kingdom [162].[40] Also in use are the PIWG benchmarks [187], used to test performance characteristics of the software, and the Hartstone benchmarks [93, 92], which test a system's ability to handle hard real-time applications.

6.2.2.1. Compile-Time Efficiency

Ada compilers require more host computer resources than compilers for most other languages, in part because the compilers are required to check for more kinds of programming errors.

Ada compilation costs can be significantly reduced if a compiler has special support tools for managing separately compiled units efficiently. These costs can also be reduced by careful use of information hiding and abstraction in the design of the software.

Ada is intended to be used not just for small, embedded systems, but also for large systems — systems that contain hundreds of thousands, even millions, of lines of code. The software in such systems may be compiled hundreds of times. Some compilers are not adequate for this level of use, due to speed or capacity limitations.

There are several reasons why Ada compilers use more computing resources than compilers for other programming languages:

- **Language capability**: Ada's inherent power and complexity generally mean additional processing is required by the compiler.
- **Earlier error detection**: Ada compilers can detect certain kinds of programming errors earlier than they would normally be detected in other languages. A type of error Ada will catch early is the classic ''apples and oranges'' problem. For example, while other languages might allow numbers representing velocity and volume to be added together, with the error eventually found during integration testing, Ada can catch such errors during compilation. Although checking for such errors has a price, a well-proven software engineering principle is that the earlier an error is detected, the cheaper it is to fix. (Of course, some errors will not be detected by a compiler for *any* language; these must still be caught during unit or integration testing.)
- **Library management**: If a compiler is not designed properly, some required error checks can consume significant resources. The key issue here is how the compiler accesses and updates an auxiliary file called the *library*. Some Ada compilers can minimize the costs of accessing and maintaining the library, and these implementations are particularly effective in their support for programming large systems. For example:

[39]The first release of the ACEC occurred in August 1988. The current ACEC (Version V3) was released in August 1992. The ACEC is being maintained by the Language Control Facility (see Section B.3) and is distributed by the Data & Analysis Center for Software (DACS), P.O. Box 120, Utica, NY 13503, Tel. (315) 734-3696.

[40]AES is not currently being distributed, as these two methods are in the process of being merged.

- Checking for inconsistencies between separately compiled units[41] can contribute to compile-time inefficiency if the compiler does not efficiently process interface information declared in separately compiled packages.
- When an interface package is changed, some compilers cannot assess the impact of the changes. As a result, large portions of the system must be recompiled unnecessarily to ensure no inconsistencies have been introduced.

- **Optimization**: Optimization refers to the process of making the generated object code more efficient for execution while insuring that the functionality of the code remains unchanged. Optimization techniques include removing redundant code, eliminating unnecessary runtime error checking, and consolidating memory usage. Since optimization of object code is especially important in embedded systems, *some*, but not all, Ada compilers devote considerable attention to generating high-quality code. This is perhaps the area of greatest progress since the first edition of this handbook [109]. Such optimization requires extensive machine analysis of the developed software, which may consume considerable resources. The use of host resources to produce code with increased target performance is an excellent tradeoff, after the software being optimized is logically correct. Many compiler products provide different levels of optimization so software development personnel can choose the most effective level of optimization for a particular stage of development.

6.2.2.2. Object Code Efficiency

Experts disagree about the time and space penalties incurred for code generated by compilers relative to optimal hand coding in assembly language. It is clear, however, that rapid progress is being made in optimized code generation and that compilers are already able to generate better code than the "average" assembly language coder [151]. Furthermore, production-quality Ada compilers have the potential to generate code that is more efficient than code generated for other languages.

Efficient object code can be generated by Ada compilers, although some current compilers may not yet achieve this goal. Just because some compilers may generate inefficient code does not imply that Ada code must be inefficient; more highly optimized compilers generate code that is as efficient as code generated for other high-order languages. Good optimizers are more the rule now than in the past. In fact, some Ada features potentially allow Ada compilers to surpass the object-code efficiencies of other languages. For example:

- More efficient machine operations can be used to manipulate Ada's composite data structures.
- Certain subprogram calls can be eliminated by putting the subprogram code ‘‘inline’’ in place of the call, thus saving time.
- Compact representations for data can be specified when compared to the compiler's normal data layout strategy, saving space.
- Certain unnecessary data compatibility checks ordinarily performed at runtime can be omitted, saving both time and space.

These features should be considered by those programs with severe object-code efficiency requirements. However, other language features, such as data compatibility checks and exceptions, can also decrease object code efficiencies. And the runtime system for any language can be a source of inefficiency as well. Some items of concern are:

[41]As part of its error checking, the compiler will ensure that a separately compiled Ada subprogram cannot be called with the wrong number of parameters. Using separately compiled units also means portions of the system can be changed without having to recompile the entire system.

- **Loading runtime software**: With some compilers, the entire runtime software is loaded, whether it is needed or not, making the executable code unnecessarily large. Thus, the ability to include only those functions that an application needs would save space. This capability should not cause any validation concerns, but may require some additional intelligence in the compiler and the linker. Many compilers already have intelligent linking, eliminating this problem.
- **Runtime customization**: Significant efficiencies can be gained if portions of the runtime system are tuned to a particular application.

Where severe time and space limitations exist, it is often necessary to invest in program-specific custom optimizations. With properly designed compilers, the process of creating, testing, and documenting such optimizations can be relatively inexpensive when compared to creating a totally new compiler. But, this process is usually not cheap and should only be undertaken when a serious need for such optimizations and sufficient resources to fund such efforts exists.

6.2.2.3. Compiler Services

The usability of a compiler is affected by the services it provides beyond the generation of code.

Compilers differ in the extent to which they provide helpful services to software engineers. The compiler services indicated below have been helpful in many efforts [121]. For more information, see [226, 179].

- **Formatted source code listing**: An indented listing revealing the structure and control flow of the source code. This capability is sometimes referred to as a *pretty printer*; these tools can often be tailored to allow each software development effort to establish its own formatting conventions.[42]
- **Assembly code listing**: A listing showing the absolute memory location of each machine instruction in the target computer.
- **Interleaved assembly code listing**: A listing showing the machine code generated for each Ada statement. (It is not always possible to produce such a listing for highly optimizing compilers.)
- **Set/use listing**: A listing showing where variables are modified and/or read.
- **Compiler error messages**: Explanations of programming errors found by the compiler. These messages are provided at the point of the error, in a consolidated listing, or both.

In addition, the following services are specifically Ada oriented:

- **Compilation order tool**: A capability that, when given a set of Ada units, determines the compilation order.
- **Automatic recompilation option**: An option whereby the compiler finds and automatically recompiles all modules requiring recompilation.
- **Recompilation analyzer**: A capability (in some cases, a separate tool) that analyzes the structure of software and indicates how much recompilation will be caused by various changes.

[42]The guidelines contained in [138] are a starting point for such conventions. Although this is the AJPO's recommended style guide, it can be tailored to allow establishment of project formatting conventions.

6.2.2.4. Support for Embedded System Requirements

Ada compilers are now being built to address the special needs of embedded systems.

Ada was intended for DoD embedded system applications, some of which impose special requirements on compilers. For example, embedded systems typically need to:

- process hardware interrupts
- place data in particular memory locations (e.g., to support memory-mapped I/O)
- specify storage-efficient representations of data
- implement code in read-only memory (ROM)
- access specialized low-level machine instructions, e.g., those that compute trigonometric functions or that allow memory checks (referred to as a built-in test — BIT) to be run when a system would otherwise be idle
- accomplish required actions at precisely specified times

Early Ada implementations did not, in general, address these specialized needs because of the pressures and demands of achieving validation. However, validated compilers that meet the special needs of embedded and mission-critical systems are now available from many vendors.

6.2.3. Compilers: Action Plan

Several factors should be considered in evaluating the quality and maturity of compilers.

Guidance regarding compiler and tool evaluation is available from a number of sources, including the Evaluation and Validation Reference System [56, 57], and the *Ada Adoption Handbook: Compiler Evaluation and Selection* [226]. Other sources include [233, 179].

The following suggestions provide general guidance for compiler acquisition:

- **Host machine resources**: Plan to support larger host machine resources (more memory and computing power) than for other languages.
- **Compiler evaluation**:
  - Specify additional criteria and tests a validated compiler must satisfy to meet program-specific needs.
  - Develop coding guidelines based on evaluation results. (Such guidelines should conform with the recommendations of [138].)
- **Compiler services**: Ensure that needed user support services are provided by the compiler.
- **Recompilation costs**: Assess vendor support for reducing the costs of compiling (and recompiling) large software systems.
- **Revalidation**: Balance the costs of revalidation (including recompiling and retesting all mission software) with the advantages a new compiler version may have in terms of reliability (fewer bugs) and improved object-code quality, as well as life-cycle support for the new compiler version.

6.3. Programming Support Environments

6.3.1. Overall Status

As software support environments will never reach their full potential for any *programming language, program managers should concentrate on tool functionality and compatibility.*

The function of a software engineering environment (SEE) is to support the development and maintenance of large-scale software systems. The SEE should provide support across the development and maintenance cycle to include specification development, test generation and testing, program management, coding, debugging, and configuration management. The goals of a SEE complement the goals of Ada, and include:

- common tools and interfaces,
- tool integration, and
- software transportability.

Program managers should primarily concentrate on:

- tool functionality
- operational compatibility between tools (including interfaces)

The following sections provide action plans for both required and optional development tools.

6.3.2. Required Tools: Action Plan

A minimal tool set is needed for use with a compiler.

As a first objective, a program manager should ensure that a good basic tool set is used. In addition to a compiler, the minimally required tools include:

- **Editor**: An interactive tool for creating documentation and source code.
- **Debugger**: Symbolic debuggers[43] for Ada are fast becoming essential development and productivity tools. The debugger and compiler work as a complementary pair, where both tools share common data structures known as symbol tables (see Figure 6-1) and are therefore best purchased from the same vendor.
- **Linker**: The linker joins together the various object modules and the runtime software into an executable module that will run on the target machine. As part of this process, it may make adjustments to the code based on the intricacies of the target processor.
- **Configuration manager**: This tool is used to achieve, at a minimum, configuration management of both documentation and code.
- **Target simulator**: While development can be accomplished without simulators, they are becoming essential development and productivity tools. They allow more development work to be done on the host machine, as opposed to repeated and often time-consuming downloading to the target.
- **Downloader**: This tool is used to download the executable code from the host machine to the target processor. Sometimes downloader software is supplied by target computer vendors.

[43]While development can be accomplished (sometimes with great difficulty) without debuggers, symbolic debuggers allow for execution of software under programmer control. They provide methods for stopping software execution at specific points and for examining and modifying data locations using source code names.

6.3.3. Optional Tools: Action Plan

Optional tools should be evaluated in light of identified needs and the tools' usefulness in supporting the software development process.

There are many other tools that are useful, but not essential, to the software development process. Several of these are briefly explained in Table 6-1. This table addresses only those tools that are directly a part of the programming process; many other tools (desktop publishing and presentation, e-mail, etc.) can also have an impact on productivity throughout the life cycle.

Before decisions are made to acquire and develop additional tools, consideration should be given to:

- **Size of the software development effort**: Small development efforts, such as those accomplished on a personal computer (PC), may not warrant additional tools. However, for large, multi-team efforts, possibly involving geographically distant participants (such as the Army Comanche helicopter and the Air Force F-22 fighter programs), tools and capabilities beyond those described in Table 6-1 will be required, as will significant management attention to the development process.
- **Documentation requirements**: User guides and online documentation should be evaluated for availability and quality.
- **Functionality, interfaces, methodologies, and compatibility**: New tools should be evaluated for compatibility with an organization's existing tools and methodologies and the Ada language [233, 105, 106, 231, 225]. An example is design tools: Are they compatible with Ada's assumptions about software organization such as packages? When public domain tools are being considered, the degree of modification required must be determined.

6.4. For More Information . . .

Details about validation, including commonly used Ada validation terms, the organizational structure for managing, coordinating, and directing the Ada validation process, steps in the process, and guidance to DoD program managers on the acquisition, use, and maintenance of Ada compilers are contained in the AJPO validation procedures [3]. Additionally, guidance for compiler evaluators is found in the *Ada Adoption Handbook: Compiler Evaluation and Selection* [226] and for tool evaluators in *Issues in Tool* Acquisition [233].

Tool	Purpose
Source-code formatter	Also known as *pretty printers*, these tools format source code using predefined coding conventions, thereby ensuring a standard visual format. Since formatting conventions may differ between software development efforts, the formatter tool should allow an organization to specify its own conventions. Sometimes pretty printing is supplied as part of compiler services, as opposed to a stand-alone tool (See Section 6.2.2.3).
Object-code analyzer	Analyzes execution paths in object code. This information can aid in testing and determining which code could be further optimized.
Static analyzer	Provides characteristic information about software such as number of lines of code, number of comments, control flow complexity, and measures of coding style. A compiler may provide this information, eliminating the need for this tool.
Dynamic analyzer	Monitors and reports the *execution* behavior of code so performance can be tuned.
Source-code cross referencer	Gives a cross-referenced table specifying where symbols in the software are declared and used, and how they are used. Some compilers provide this information directly. A more advanced tool allows a user to "browse" online, moving automatically from the place where a name is used to the place where it is defined.
Syntax-directed editors	Assists in creating source code by analyzing and checking for many types of errors as the source code is entered. The advantage of these advanced editors (also known as language-sensitive editors) is identifying certain types of errors early, without having to process the code through a compiler.
DoD-STD-2167A documentation generators	Supports the generation of documentation according to the formats of DoD-STD-2167A data item descriptions (DIDs). Often employed in conjunction with program design language (PDL) tools and/or graphical programming tools.
Graphical programming tools	Allows software engineers to visualize the design process and the resulting system design as it develops. These tools are relatively new, but progress in this area has been encouraging.
PDL processor	Supports the processing and documentation of software designs.
Test manager	Helps organize and execute software tests, including comparing test results from previous runs.
Module manager	Accepts directions about how to build a system baseline; determines which modules in a system have been changed and which modules are potentially affected by the changes.

Table 6-1: Optional Tools

7. Ada Applicability

7.1. Scenarios for Ada Use

Ada can be used today for applications that range from information systems to embedded, time-critical applications and that run on machines from microprocessors to much larger computers. For severely time-critical (i.e., hard real-time) applications on resource-constrained machines, special care is required in selecting appropriate compilers and tools.

Given the wide spectrum of DoD applications and the wide variety of processors in use, any evaluation of Ada applicability must address:

- requirements of the application domain, such as:
  - type of computers used as the target processors
  - processing requirements (real-time, etc.)
  - resource constraints (memory, peripheral equipment, etc.)
  - physical environments (ground based, airborne, spaceborne, etc.)
- maturity of software technologies
- capabilities of candidate target processors
- maturity and capability of development organizations

The following paragraphs briefly survey the current situation and answer the question, "How and where can Ada be used?"

Low Risk:

- Applications that are not time critical and run on machines with medium to large memory capacity. Such applications include software tools, support software, simulations, and many ground-based applications.
- Time-critical, ground-based applications running on machines with medium to large memory capacity going into *immediate* development.

Medium Risk:

- Large command, control, communications, and intelligence (C^3I) projects involving complex interfaces and loosely coupled distributed processing or databases. (See Section 8.2 for more on the topic of interfacing to commercial software packages.)
- Time-critical, embedded (i.e., airborne/spaceborne) applications going into engineering and management development (EMD) in the next 2 to 4 years. This situation can be made somewhat less risky by selecting target processors that provide more Ada support.
- Current (e.g., developing code) embedded time-critical applications, such as airborne/spaceborne applications using embedded avionics processors

In any of these medium-risk scenarios, program management must make judgments about Ada technology compared to the schedule of the actual program. Consideration might be given to including (and funding) risk reduction and/or technology insertion phases in the program, especially for those projects making first use of Ada.

High Risk: Massively parallel systems and certain distributed systems (See Section 7.5).

Although there are risks in developing software-intensive systems, Ada has proven to be the best language available in many cases (such as [151]), and many of the risks are *not* language specific, but are driven by issues of complexity in interfaces or time criticality. Ada provides a way to enable good software engineering and management principles to be applied in ameliorating these risks.

7.2. Ada and System Engineering

System engineering tradeoffs are increasingly reflecting the importance of software engineering issues in developing systems today. Managers should develop action plans to address these tradeoffs as part of their risk management activities.

Successful adoption of Ada depends not only on the maturity of compilers and software tools, but also on system engineering analysis of the program schedule, hardware choices, performance requirements, and supportability issues; these should be mapped against Ada capabilities. The program manager *must* insure that software tools are available for the processors selected; joint selection of the target computer and support software is the optimal approach. In the following subsections, various system engineering situations are examined, and possible action plans suggested.

7.2.1. Absence of Compiler: Action Plan

Several options are available if no Ada compiler exists for the selected hardware.

An action plan for evaluating and selecting a compiler was presented in Section 6.2.3. If no Ada compiler exists for the selected target computer:

1. Select another target computer for which Ada compilers already exist, if possible. The official AJPO list (October 1992) contains 270 validated *base* Ada compilers of the 501 *total* validated Ada compilers. Because Ada compilers are available for a wide variety of computers, some choice is possible in selecting compilers for various host-target combinations, [44] and it should *not* be necessary to build a completely new compiler.

2. If program cost and schedule allow, have a code generator developed for the target processor, building on an existing compiler. The current trend is to use commercially developed products because of their diversity, lower cost, and quality, rather than developing a new compiler.

   This process of retargeting (extending a compiler to generate code for a new target computer) is also known as building a new code generator or compiler back end.[45] If the manufacturer of the target processor provides a tool set (linker, loader, debugger, assembler, etc.), determine what extensions or additional capabilities may be required.

[44]The current list of validated compilers, including host and target information, is available in printed and electronic form from the Ada Information Clearinghouse and is also published in the Ada-JOVIAL Newsletter. See Appendices A and B for further information.

[45]A compiler consists of at least two parts — the front end and the back end. The front end, which is also called the machine-independent portion, contains the components that are independent of the target computer and can be used in common for many target computers. The front end generally will not have to be modified during the retargeting process. The back end contains the components that depend on the characteristics of the target computer and therefore must be designed specifically for each target computer.

These activities are likely to require the use of program funds; there is generally no DoD-wide funding for these efforts. Retargeting development time should range from 9 to 15 months and cost from $750,000 to $1,750,000. These estimates may vary depending on a number of factors, including:

- the requirements and technical objectives of the effort
- the compiler technology used
- the difficulty of the code generation task
- the maturity of the compiler developer
- access to the developer for resolving problems
- economic factors such as the competitive position of the developers bidding for the work and the potential market for the compiler. (For example, a compiler for a target processor used extensively would most likely cost less to develop than a compiler for an aged or proprietary processor that has limited demand in the marketplace.)

Additional time should be planned to *mature* the newly retargeted compiler. The time required for new or modified tools to reach an acceptable level of maturity should also be considered, as discussed in Section 4.1.3.

3. If these alternatives are not possible, consider the risk and cost effectiveness of using Ada. Consider filing a waiver requesting approval to use another approved language if an Ada development is shown not to be cost effective.

7.2.2. Object-Code Efficiency: Action Plan

Concerns about object-code efficiency can be addressed by a number of alternative actions.

Where the code generated by an Ada compiler might be an issue, either due to its performance, quality, or efficiency, there are several alternatives:

1. Re-analyze performance requirements and then plan and conduct an appropriate evaluation effort. (See Section 6.2.2 and [226, 227, 93, 204].)
2. After sound engineering analysis, avoid the use of selected Ada language features that cause or are likely to cause difficulty.
3. Obtain a compiler that produces better quality code, including optimizations, for the selected processor.
4. Selectively use assembly language. Based on sound engineering analysis of system throughput bottlenecks, code parts of the system in assembly language. Note that the compiler in use *must* support either machine code inserts or *pragma*[46] INTERFACE for assembly language. (For additional information about mixing languages, see Section 8.4.1.)
5. Alternatively, fund modifications to an existing compiler to meet program-specific performance or resource constraints or the development of a new compiler that produces better quality code for the selected processor. Such activities include improving the generated object code, tailoring the runtime software, and adding additional debugging capabilities. Work with the compiler vendor to provide a series of benchmarks specific to the application that the compiler *must handle with adequate performance* as a basis for accepting the modified compiler.

[46]Pragmas are used to convey information to the compiler, for example, as an instruction to the compiler to perform some special action such as compiler optimization or to interface to software written in other languages. There are two kinds of language pragmas: those that are predefined in the language and those that are defined by the compiler vendor.

6. Add resources (such as additional memory) to the selected target processor. However, adding memory *may* require modifications and enhancements to the existing support tool set, and may have an impact on weight, cooling, power, and other requirements.
7. Upgrade to a more powerful processor. While this is in principle the best solution for making a more evolvable system, there may be issues regarding processor qualification to military standards and supportability after the system has been deployed. Also, upgrading to a new processor *may* require modifications and enhancements to the existing support tool set, and may have an impact on weight, cooling, power, and other requirements.
8. Add additional processor(s) and decompose the software onto the processors. Concerns here are the additional overhead and complexity of the protocols (more software) that will be needed for the processors to exchange information. This processing load is difficult to quantify and is further affected by system requirements for fault tolerance, multi-level security, and the buses used to connect the processors. Adding additional processors may also cause weight, cooling, power, and reliability problems.

7.3. Ada and Embedded Processors

A wide variety of embedded processors exist. For many of these processors, Ada is an appropriate language choice. However, it is generally not cost effective to build or use any *high-order language, including Ada, on processors with unusual architectures, highly restricted instruction sets, or very specialized purposes.*

Using *any* high-order language (HOL), including Ada, may not be cost effective or even possible for certain classes of processors since all processors have varied capabilities. Some processors are designed and optimized for single purposes, while others are intended for general-purpose use. This orientation directly affects which programming language(s) can be used on a particular processor. Some recently developed processors (that possess large storage capabilities and instructions for conveniently accessing this storage) provide a more attractive target for high-order languages (including Ada) than do older, more resource-constrained processors.

Table 7-1 assesses the applicability of Ada on various processors based on the following factors:

- **Tool development**: Effort and cost required in developing and maintaining compilers and support tools.
- **Quality of the resulting product**: Specifically, what kind of code is generated? After the code is generated, will additional application and project-specific optimization be needed?
- **Effect on software developers**: Does the underlying hardware help or hinder the application software developer? Do attributes of the target processor add complexity, lower productivity, or increase the possibility of error?

7.3.1. Non-Standard Architectures: Action Plan

When considering a non-standard architecture, several factors should be addressed in addition to the evaluation of available Ada compilers.

While innovative approaches to processor and memory configurations are always encouraged, prior to adoption:

1. Consider potential difficulties in hardware evolution as the system goes through its operational life cycle.

2. Evaluate the impact on required software support. The costs of supporting unique or special versions of software development tools for non-standard memory configurations over the system life cycle can be considerable. The program manager should determine what special tools (if any) will be necessary and what modifications to existing tools will be required, and balance this against the advantages of the proposed hardware configuration.
3. Consider the potential impact on future competition if the system design is locked into non-standard or proprietary architectures.

Category	Description/Explanation	Examples
Applicable	General-purpose 32-bit processors characterized by large linear address space and rich instruction set.	DEC Vax family, Intel 80486, 80386, 80960, Motorola 68040, 68030, and 68020, SPARC, MIPS processors
Applicable with Minor Reservations	General-purpose 16-bit processors and resource constrained 32-bit processors. Extended memory and distributed machine applications might provide minor problems.	Intel 80286, Intel 80186, Motorola 68010 and 68000, MIL-STD-1750A, AN/UYK-43
Applicable with Reservations	Older, general-purpose 16-bit processors, usually characterized by hardware memory constraints. Expected problems are extended memory and distributed machine applications.	Intel 8086, MIL-STD-1750A, AN/UYK-44, AN/AYK-14
Limited Applicability	Special-purpose machines. Ada compilers are available for some digital signal processors and special-purpose processors.	Digital signal processors (such as TI C30 DSP or Motorola 96002), array processors
Not Applicable	General-purpose 8-bit processors.	Intel 8088, 8051, 8031, Motorola 6809

Table 7-1: Ada on Various Processors

7.4. Ada for Real-Time Systems

Ada has been shown to be effective for the implementation of real-time systems. New design paradigms, such as rate monotonic analysis, have allowed Ada to address the complexity in real-time systems design.

Concerns have been raised about Ada's usefulness in programming real-time systems.[47] These concerns, which are somewhat anachronistic since some compilers have solved these problems, fall into two areas:

1. **Efficiency**
   - Some Ada implementations do not handle interrupts with acceptable efficiency. However, some implementations can handle interrupts with an efficiency equal to assembly code. (Usually this approach requires the use of pragmas.)

[47]Real-time refers to computer processing in which the timeliness of the functions is as important as the correctness of the functions. Real-time systems control, direct, or influence the outcome of an activity or process in response to the data it receives from the activity or process. Examples of real-time systems are process control, target acquisition and tracking, and computer-aided navigation.

- Some Ada implementations take too long to switch from running one task to another.[48] However, some implementations have been very efficient in context switching performance or switching between tasks.
- Code generated by some Ada compilers is less efficient in time and space than that generated by compilers for other languages. Code generated by Ada compilers can be as efficient or more efficient than code generated by compilers for other languages. (See Section 6.2.2.2.)

2. **Timing Control**
   - When a task needs to execute at a periodic rate, it appears to be difficult to control the rate with sufficient accuracy. Hartstone benchmark results [92] have proven that many Ada compilation systems can reliably meet hard deadlines.
   - Ada tasks are scheduled for execution under control of runtime software (a *scheduler* or *executive*) written by the compiler vendor. If this software does not meet the needs of a particular real-time application, it may be difficult or costly to modify. Most compiler vendors will tailor their runtime systems or make available source code for their runtime systems for an appropriate licensing fee.
   - It can be difficult to analyze the timing behavior of a system of Ada tasks, and the behavior can depend significantly on runtime software that is not under the direct control of application programmers. Appropriate analysis and benchmarking technologies, such as rate monotic analysis [204] and Hartstone [93], allow the timing behavior of a system comprised of Ada tasks to be analyzed.

These concerns generally apply to some implementations of compilers and runtime systems, not to the language, and do not mean that Ada cannot be used for real-time systems. Real-time systems have been written in FORTRAN and JOVIAL, languages that do not support concurrent processing directly, as well as in Ada.

7.4.1. Real-Time Systems: Action Plan

Ada has been used successfully in real-time systems. Several actions can be taken to aid in using Ada for real-time systems.

To ensure the successful use of Ada for real-time systems:

1. Assess the performance and timing requirements of the proposed system. This may require prototyping and benchmarking, as well as analysis. Rate monotonic analysis may be an appropriate technique for use in designing many real-time systems [204].
2. Check that the proposed compiler efficiently supports real-time features needed, such as interrupt handling and context switching. If interrupt handling is not efficient, consider using assembly language for interrupt handling (which is a small portion of a system's overall code). If context switching is not efficient, consider selecting another compiler after a thorough evaluation.
3. Design the system correctly. Analyze the required execution threads (or processing flows) for each stimulus (or input) through to the response (or output) and minimize the task switches on critical threads. A timing analysis can indicate when to combine Ada tasks, as an appropriate number of Ada tasks leads to an efficient design. Rate monotonic analysis [189] may be an appropriate technique for performing timing analyses and evaluating system design.

[48] *Tasks* are Ada's way of achieving concurrent processing.

7.5. Ada for Distributed Systems

As with other uses of Ada, Ada should provide improved maintainability and reduced long-term costs when used for developing distributed systems.

Although no currently approved DoD language was designed specifically for developing distributed systems, Ada can be used in this domain [33, 60, 88, 170, 31, 114, 67, 19, 63, 15]. One way of using Ada in distributed systems is to write independent Ada programs that execute on each node of the system. The software resident in the various nodes communicate by a message-handling protocol implemented outside the Ada language. This loosely coupled implementation is typical of many existing distributed systems, and Ada can be used in the same way. The advantages of using Ada instead of some other language are improved maintainability and reduced long-term costs.

A second kind of distributed system is a tightly coupled system, where multiple processors share an address space. The use of Ada for these systems is possible, but brings increased risk.

A third kind of distributed system is a massively parallel system for which Ada compilers are not yet available, although some vendors are working toward those capabilities [166].

8. System Design and Implementation Issues

8.1. Designing for Portability and Reuse

Ada's standardization and its language features mean that Ada is especially suitable for promoting software reuse. However, Ada does not automatically make software reusable. Rather, Ada enables development of portable and reusable software.

8.1.1. Portability

Portability of software across computing platforms can be enhanced through effective use of Ada.

Portability is the extent to which software can be moved *without modification* between different computing platforms (hardware, operating system). Portability can be either at the level of source code (which must be recompiled on the new platform) or executable code (which can be executed on the new platform). Portability has traditionally been a significant problem due to programming language (and dialect) proliferation; some modification seems to almost *always* be required when porting.

The source of the software, the development environment, execution environment, and application domain have been the key components in assessing the portability of software. When porting software, the reasonableness of the port must be considered. For example, porting a piece of code that implements a fairly generic function is often straightforward, but porting software that is tightly coupled with particular hardware or operating system constraints is usually much more difficult.

As opposed to other languages, a major benefit of Ada is that Ada is standardized, and compiler validation has effectively limited the evolution of non-standard language dialects. Thus, Ada portability compares very favorably to other languages because of language standardization, the modularity constructs of Ada (packages), and its abstraction capabilities (e.g., types, attributes), which encourage isolation and localization of machine dependencies. Ada's language standardization and compiler validation are important contributions.

But compiler validation (compliance with the ACVC) and language features do *not* guarantee portability. Portability can be compromised by:

- tradeoffs made to maximize performance
- environment-specific features (operating systems services, user interfaces, database management system (DBMS), etc.) used
- implementation-dependent features (data representations, capacity limitations)
- poor attention to portability during system design

Portability does not occur automatically through the use of Ada. It must be planned, designed, and coded into the software. However, portability can provide substantial benefits. In one MIS application, the man-machine interface software has been ported to six different targets with minimal impact [214].

8.1.2. Reuse

Ada provides features that can enhance reuse of software architectures and components.

Ada has helped overcome some of the historical inhibitors to reuse. Ada's language standardization and unique facilities such as packages, generics, and separate compilation have provided a catalyst for reuse. Ada allows interfaces to be precisely defined so functions performed by a software unit are obvious, precise, and regulated. These capabilities, if properly used, will facilitate the reuse of software components. However, Ada does not *automatically* make software reusable; rather, facilities are available to design a component for reuse. Indeed, programming for reuse may require a higher level of effort than normal programming [21].

There are numerous successful instances of reuse-based development that have resulted in significant cost savings and quality improvement for the organizations involved. These range from commercial process control applications to air traffic control systems and advanced real-time command and control systems [184, 228]. The French firm Thomson CSF reports 50 percent to 70 percent and higher reuse rates in developing numerous air traffic control systems, entirely in Ada, for their various international clients [228]. Ada reuse is continuing to grow [166].

8.1.3. Designing for Portability and Reuse: Action Plan

Ada provides capabilities that can be used to support portability and reuse concepts.

To achieve portability or develop reusable software, projects must:

1. State portability or reuse as a design objective.
2. Plan appropriate time/budget (portable/reusable systems may cost more to develop initially).
3. Localize implementation dependencies (collect system-dependent objects/operations into as few packages as possible).
4. Design for change. Group parts of the system that are expected to change together using Ada packages.
5. Identify and enforce use of well-considered Ada features (e.g., *attributes*) and style guidelines.[49]

Whenever possible, design every system as if it will have to be ported to a variety of configurations through which the system may evolve over a 20- to 30-year life cycle, given fixed budget, schedule, and resource constraints. This is not far-fetched, as systems developed today will be in operation 20 years from now, and hardware advances are occurring almost every year. Following modern software engineering practice, portable software localizes dependencies on the target machine and insulates the rest of the software from those dependencies. These practices also improve the quality of the software.

Portability and reuse are design goals that management should plan for by allocating sufficient resources (time and money) to development efforts to plan for and meet these goals. Portability and reusability are not free, but should be given serious consideration in managing software efforts.

[49]See also *Portability and Style in Ada* [178], *Ada Quality and Style Guide* [138], and [96].

8.2. Using Ada with Other Standards and Protocols

Ada interfaces to other systems and standards are available. New implementations of interfaces (or bindings*) are continuing to emerge. This is a direction of the future; while early concerns regarding Ada use dealt primarily with language implementations, attention is now turning to successful integration of Ada software in overall systems environments.*

As Ada use becomes more common in software-dependent systems, and as requirements grow, system developers will be faced with both the benefits and problems of adopting more disciplined and integrated approaches to software engineering. It is important that Ada applications be able to effectively use whatever resources are needed to produce a complete system. These resources could include operating systems or communications services, database management systems, graphics packages or user interface management systems, or interfaces to COTS software or other existing systems. These resources may be formal standards, de facto standards, or commercial products that have become widely available.

Using these resources from an Ada application will frequently require that an Ada binding to these standards and protocols be developed. Each of these bindings is a definition of a set of services that can be provided to an application and an interface through which those services can be accessed. For an Ada program to use these services, a definition of the interface (or *binding*) must be expressed in terms of the Ada language. The application developer still needs to have available an implementation of that binding to build Ada applications.

Sometimes a binding effort may take place as part of a formal standardization program — with public input and representation from both the Ada community and the community most interested in the standard. At other times, a particular organization or company may implement a binding only for a particular product, Ada compiler, application, etc.

Foremost among these issues is the reality that Ada is not often the first choice for defining and implementing bindings. A troubling issue is that an Ada binding may be issued later than a C binding and the implementation of the Ada binding may even be later.

Program managers need to be very careful that they do not over-constrain their contractors by requiring "Ada for the sake of Ada" and ignoring sensible, thorough, complete life-cycle analyses that effectively demonstrate when reuse, standards, COTS, or languages other than Ada would provide a more cost-effective life-cycle solution.

Numerous Ada bindings exist (e.g., POSIX, SQL, X Window, GKS, Motif, PHIGS) and more are being developed. The AJPO has completed a survey of available bindings [133]. Reasonable future expectations on bindings and standards issues include continued and increased availability of bindings and secondary standards for Ada.

The following sections address three commonly discussed bindings: POSIX, SQL, and the various graphics and window system bindings.

8.2.1. POSIX Ada Bindings

Ada bindings to POSIX have been standardized, and Ada bindings to POSIX real-time extensions are being developed.

The Portable Operating System Interface for Computer Environments (POSIX) is an international standard for operating-system interfaces [149]. POSIX is defined as a full set of standards, including a base standard (ISO/IEC 9945-1:1990/IEEE Std. 1003.1-1990) and several functional extension standards covering such topics as real-time (IEEE Std 1003.4/4a), shell and utilities, and supercomputing. The base POSIX interface standard is derived from a variety of UNIX specifications and implementations.

The Ada POSIX Standards Working Group (IEEE 1003.5)[50] is developing two Ada interface standards:

1. **Ada interface to base POSIX:** The Ada binding to the POSIX system services has been approved as an IEEE Standard (IEEE Std. P1003.5-1992).
2. **Real-time interfaces:** A follow-on project, IEEE P1003.20, is defining Ada language interfaces to the P1003.4/4a real-time POSIX extensions. Work commenced in October 1991.

The base POSIX Ada binding document defines the set of standard Ada 83 package specifications and provides English text describing the semantics of those packages. This binding definition covers the set of system services defined in POSIX 1003.1, including files, directories, processes, and signals. It also includes an interpretation of some Ada features (e.g., TEXT_IO) in a POSIX context. It does not cover facilities covered in other POSIX documents (e.g., real-time, security, system administration, and networking). Several vendors are planning implementations of the Ada binding. In general, these implementations will be compiler specific, as some features of the binding require cooperation from the compiler and its runtime.

8.2.2. SQL Bindings

Several approaches to using Ada with the SQL interface to database management systems have resulted in emerging standards.

The Structured Query Language (SQL) is an interface to database management systems. There has been much interest in Ada bindings to SQL, as only bindings for the COBOL, FORTRAN, Pascal, and PL/1 languages were included in the SQL standard [14]. The Ada binding is in the current ISO SQL2 standard (ISO 9075:1992).

The SEI has investigated the problem of binding the Ada programming language with the SQL database language [95]. The solution to this problem was the specification of the SQL Ada Module Extension (SAME), an architecture for Ada SQL applications that emphasizes software engineering principles [115]. The SQL Ada Module Description Language (SAMeDL) [52] facilitates the construction of Ada SQL applications having the SAME architecture. The ISO Ada working group is preparing the SAMeDL for standardization [213]. This process is due to be completed in 1993.

[50]See Appendix B.2 for contact information.

8.2.3. Graphics and Window System Bindings

Ada bindings have been implemented for a number of graphics or windowing systems standards that are in widespread use.

In implementing graphical user interfaces (GUI), several graphics or windowing system standards are often used. These include Graphical Kernel System (GKS), Programmer's Hierarchical Interactive Graphics System (PHIGS), and the X Window System. Various Ada bindings have been implemented for these standards.

For example, the Ada binding to GKS, which is a standard library of subroutines for an application programmer to incorporate within a program to produce and manipulate graphical images, is an ANSI/ISO standard (ANSI document #X3.124.3/ISO 8651-3). The Ada binding to PHIGS, which is a graphics system designed to support computer graphics applications that are highly dynamic and interactive, is also an ANSI/ISO standard (ANSI/ISO 9593-3 - PHIGS/Ada binding). Work is underway to develop a binding to the enhanced PHIGS — PHIGS PLUS. The PHIGS PLUS/Ada binding, when approved, will be published as an amendment to the PHIGS/Ada binding.

The X Window System is a network-transparent window system. It supports one or more screens containing overlapping windows or subwindows. X Window systems are typically used with either the OSF Motif (Open Software Foundation/Motif) or the Open Look graphical user interfaces. Numerous Ada bindings or Ada-based user interface toolkits interfacing to X Window or to its toolkit libraries have been implemented.

8.3. Using Special-Purpose Languages

Ada is better for some applications than for others. In some cases, special-purpose languages may still be appropriate for specific applications.

Special-purpose (or problem-oriented) programming languages have historically been developed to support highly specialized applications. As such, there is a large, mature base of software in these languages and application areas that may take several years for Ada to replace. The languages (and application areas) considered below are:

- **ATLAS** for automatic test equipment applications
- **Simulation languages** such as SIMSCRIPT and GPSS
- **Artificial Intelligence languages** such as LISP
- **''Fourth-generation'' languages** for database and interactive applications

8.3.1. ATLAS

Although DoD policy had specified that Ada is preferred (but not required) as the language to be used for hardware unit under test (UUT) equipment [78], current DoD policy [79] authorizes the use of ATLAS in automatic test equipment.

ATLAS [131] is a programming language designed specifically for use in controlling test equipment. As such it is an acceptable and often appropriate choice for use in automatic test equipment. ATLAS is a DoD-approved language [77] for programming automatic test equipment (ATE). It contains special features that allow the direct expression of wave forms, timing requirements, and stimulation patterns, all of which are critical to the development of this type of software.

Although the functionality of ATLAS could be supported in Ada (by writing Ada subprograms that have functional capabilities similar to those provided by ATLAS commands), much code provided directly by ATLAS would have to be specially written if Ada were used. While much of this software could be reused in subsequent applications, the initial development cost would be significantly higher if the required software were not already available in Ada. In addition, software written in ATLAS may be more readable by test engineers than equivalent software written in Ada, simply because ATLAS has special language constructs that are particularly suited to expressing ATE algorithms.

On the other hand, Ada's superior capabilities for giving large software systems an understandable structure make it a preferable choice for large, complex ATE projects. There is ongoing work in the IEEE Ada-Based Environment for Test (ABET) standards development arena that will take advantage of Ada programming and implementation features while retaining the high-level, Unit Under Test (UUT) test specification aspects of ATLAS [165, 169]. This standard (IEEE 1226) is planned for release in 1994.

8.3.2. Simulation Languages: SIMSCRIPT and GPSS

For simulation programming, it may be preferable to use one of the special-purpose simulation languages; nevertheless, simulation software is being written effectively in Ada.

SIMSCRIPT and GPSS are examples of special-purpose languages for programming simulations. For example, these languages allow a software developer to sample from various probability distributions, specify when certain events of interest to the simulation are to occur, and indicate what data are to be gathered when the events occur. They also have special features that help in reporting results.

Effective simulation software has been written in Ada. Ada has been used extensively in the flight simulator domain.[51] Since Ada has no language constructs or predefined subprograms specially designed for simulation programming, simulation developers will initially have to write more code to provide the same special-purpose simulation language facilities in Ada. Ada software can be written to support simulation programming, and many of Ada's initial applications have been in the simulation arena. Once written, these routines can be used by future simulation efforts.

8.3.3. Artificial Intelligence Languages: LISP

Once an artificial intelligence (AI) technique is well understood and is to be engineered into an application, it may be appropriate to use Ada as the language for implementing the operational application. For AI research, LISP is more appropriate.

LISP is the language of choice for artificial intelligence (AI) research. Like Ada (and unlike ATLAS, SIMSCRIPT, or GPSS), LISP is a general-purpose language. LISP programming is typically done in a rich programming support environment; the programming support tools (which are written in LISP) are a strong reason for continued interest in LISP.

LISP is used for AI research applications because LISP software is easy to change. Ease of change is important in AI research, where the objective is usually to discover how to make a computer perform a task that is not well understood. LISP offers a high degree of flexibility to small teams of skilled

[51]See [157] for one such example that has had wide impact in this community.

developers working on prototypes, but that very flexibility may be inappropriate for a large team of programmers facing a significant engineering activity.

Ada supports a different view of software development, which says that changes can be hard to implement correctly and should be done using a language that helps detect inconsistencies in a changed system, and supports administrative control over the change process. This viewpoint is appropriate for large, long-lived systems, which may have several versions to be managed and maintained.

These viewpoints are less clear when moving from research to operational applications. Although LISP has been used extensively for AI research, if the result of some AI research is to be used in an operational system, LISP may no longer be the most appropriate language. For example, making software easy to change has a price — LISP software is typically less efficient in the use of time and space than is Ada. In addition, because LISP is easy to change, it is easy to change incorrectly; such errors are usually easy to fix during the research phase but difficult to fix when a system is widely distributed and in operational use.

Depending on a sound engineering study of an application, LISP may be too inefficient for operational use. If so, the information developed during the research phase can be used to design a more efficient (albeit less easily changed) system that can be used operationally. In such cases, it will probably be better to implement the operational system using Ada. On the other hand, LISP may continue to be an appropriate choice for an operational system if:

- time and space efficiency are not critical or the target computer is specially designed to execute LISP code efficiently;
- functional changes will be required frequently (e.g., at least monthly), but the nature of the required changes cannot be predicted easily; and
- sufficient safeguards are taken to ensure that changes are made correctly or to minimize the effect of an incorrect change.

8.3.4. Fourth-Generation Languages

Fourth-generation programming languages are cost effective when used appropriately. Their main drawbacks are limited domain of usability (they cannot satisfy the needs of some applications), slower performance, and possible dependence on a single vendor.

Fourth-generation languages (4GL), which include application-specific database languages, spreadsheets, and non-procedural languages, are not direct descendants of, and not necessarily improvements over, third-generation, general-purpose languages. They might better be called program generators or application-specific tools and are designed for a limited domain of use.

If an application is well matched to a fourth-generation language, the cost of realizing the application can be significantly less expensive than programming it in a general-purpose language such as Ada. For example, spread sheets are routinely used to develop in hours applications that would require days in a general-purpose language like Ada.

While fourth-generation languages offer significant advantages in productivity and rapid prototyping, there are some significant disadvantages to consider:

- **Limited domain of applicability**: Fourth-generation languages are quite effective when used for appropriate application domains, but the area of appropriate use is bounded — if some part of an application lies outside the domain of a fourth-generation language, serious performance penalties may result, or the language may not be usable at all. A project might initially have great success using a fourth-generation language, only to discover later that certain requirements cannot be met successfully and a completely new approach is required. In addition, fourth-generation languages to date have been most effective when applied to small projects; large projects have presented additional difficulties.
- **Performance**: An application using a fourth-generation language usually runs slower than an equivalent solution written in a high-order language. This performance degradation may be an acceptable tradeoff in some applications, given the increase in programming productivity, but fourth-generation languages are usually not well suited to applications that stretch the performance capabilities of a system.
- **Vendor dependence or lack of control**: Fourth-generation languages tend to be commercial products and are thus highly susceptible to the pressures of the marketplace. If an application is meant to have an extended life cycle, the product may change and become incompatible or even disappear.

Chances of success with a fourth-generation language are good if the requirements fall within the domain of the particular fourth-generation language. If this criterion is not met, there is considerable risk that the effort will result in failure.

8.4. Mixing Ada with Other Languages

Mixing Ada with other languages is possible, provided consideration is given to technical and management issues. On the other hand, ‘‘automatic’’ translation of code written in other languages should be approached cautiously.

The question of mixing languages is important to managers of both new developments and existing operational systems (post-deployment software support). Among the questions of interest are:

- Should Ada be used in the next upgrade?
- Is it appropriate to develop and maintain a system partly in Ada and partly in some other language, thereby creating a hybrid system?

The following criteria can be used to answer these questions:

- Can the Ada code interface with code written in another language?
- Is the replaced portion relatively independent of the rest of the system?
- Is much of the system being replaced?
- If a hybrid system seems inappropriate, should the system be rewritten entirely in Ada, even though large portions of code would not otherwise have to be changed?

8.4.1. Interfacing Ada Code with Other Languages

When mixing Ada with other languages, several technical issues should be addressed.

Among the issues to be considered in attempting to interface Ada with other languages are:

- Can the Ada code call subroutines written in other languages? Alternatively, does the new Ada code have to be called from the rest of the system? For example, can a COBOL or JOVIAL compiler generate code to call Ada subprograms?

- What kind of data have to be processed by both the Ada code and the rest of the system? Can the data be represented in a way that is acceptable to both languages?
- How much redundant or unused runtime support software will be present in the completed system? Can the runtimes for the languages co-exist efficiently?

The following subsections present further details.

8.4.1.1. Subroutines Not Written in Ada

Efficiently interfacing Ada with another language requires the ability to call subroutines written in other languages.

Subroutines written in COBOL, JOVIAL, assembly, or some other language can be called from the Ada software *if* an Ada compiler supports the *pragma* INTERFACE for those languages. If this pragma is not supported for the language and compiler[52] of interest, the cost and schedule implications of obtaining support should be determined. Similarly, if the existing code must call Ada subprograms, modifications to the compiler for the other language may also be necessary.

However, some vendors have implemented multi-language integrated environments whereby any of their supported languages (Ada, JOVIAL, FORTRAN, etc.) can call subprograms in any language.

Another strategy is providing explicit *implementation-dependent* import/export capabilities. From any vendor-supported language, one can access subprograms and (in some cases) data objects from programs in other languages. There is a growing request for a more standard interface to achieve this.

8.4.1.2. Compatible Data Representation

Compatibility of data representation is a critical although often overlooked factor in interfacing with another language. Incompatibility can lead to inefficient code or difficulties in implementing an interface.

Another requirement for implementing a hybrid system is assuring the compatibility of data representation between Ada and the other language.[53] The data interface between the languages must be specified and controlled, just as it is between processors and buses. Key questions program managers must answer include:

- What data will be interchanged between the Ada and non-Ada portions of the system?
- Will the storage layout for these data be the same for both languages? If not, can it be made the same?[54]
- If the storage layout cannot be made identical, how will efficiency be affected?

---

[52]The interface is created to a *specific* compiler — for example, a specific FORTRAN compiler, not to just any FORTRAN compiler.

[53]For example, arrays in FORTRAN are typically stored differently from arrays in Ada. So, although a FORTRAN matrix inversion routine may be called from Ada, the FORTRAN language will view the matrix as though its data are arranged differently from the Ada view and will produce incorrect results. (This kind of incompatibility is not uncommon between languages.) On the other hand, if the Ada implementation is designed to support interaction with FORTRAN software, it may store arrays in the manner required by FORTRAN, or it may give a means to specify how data are represented so no problem will arise.

[54]While the Ada language provides special features (representation clauses and implementation-dependent pragmas) that allow software developers to specify data storage layouts in detail, at present not all Ada compilers adequately support these facilities.

8.4.1.3. Redundant Runtime Support

Using two languages may require examining how the functionally redundant runtime software environments will co-exist.

Runtime support for a language is the code needed to implement its complex features. If a system contains code written in two or more languages, the runtime support code for both languages will generally be needed. For example, suppose a system is written in COBOL, which has extensive I/O support facilities. Suppose the unmodified portion of the original system continues to use COBOL's I/O facilities, and the replaced portion uses Ada's I/O facilities. Runtime support code will be needed for both the Ada and COBOL portions of the system, which can be unnecessarily expensive in space. If, instead, the modifications were designed so that all I/O processing were done by the COBOL code, then Ada's I/O runtime support code need not be loaded. The savings in space could be significant in a resource-limited system.

Although it may not be necessary to load Ada's runtime support for I/O, some less mature compilers may load it anyway. The extent to which a compiler loads only the needed runtime support software is an important factor in deciding whether it is feasible or cost effective to write just a portion of a system in Ada. Only a detailed analysis will provide answers to these important factors.

Key questions regarding runtime support include:

- Can the runtime environments co-exist? Various languages assume different system memory models or have different paradigms regarding the scheduling of tasks. This type of incompatibility may cause unpredictable system behavior.
- How much runtime support code is associated with the use of particular Ada language features?
- Is runtime support software loaded whether or not it is actually needed?
- Is some of the runtime support essentially the same for both Ada and the other language? If an overlap is significant, is it necessary to use the associated Ada features?

8.4.2. Isolating Subsystems

To minimize language incompatibility, the portion of the system that is written in one language should be relatively independent of the portion written in another language.

It may be appropriate to write a new portion of a system in Ada if the new portion is relatively isolated from the unmodified portion. Two examples where Ada might be successfully introduced are:

- **DBMS/file system interfaces**: Suppose the unmodified portion of a system is a database management system (DBMS) written in COBOL or purchased from a vendor (so its source code is not available). If the remainder of the system consists of software that accesses the DBMS and processes the results of DBMS queries, then calls to the DBMS can probably be written as easily in Ada as in any other language.
- **Distributed systems**: On a distributed system, it should be possible to replace the code that exists on the individual processor node(s) as long as the system-wide interfaces are maintained.

8.4.3. Replacing the Whole System

Possible indicators that a system should be totally reimplemented in Ada include continual trouble reports, significant software change, or new target hardware.

If a system is operating smoothly, replacing that system just for the sake of having it in Ada is generally *not* recommended. However, there are several rules of thumb that may indicate the need for a system replacement. Assuming the availability of tools for the target processor, Ada should be the replacement language if any of the following situations exist:

- **Poor overall system reliability**: The existence of unresolvable trouble reports, significant difficulties in making corrections or enhancements, or an increase in new bug reports after correction/enhancement activity may indicate fundamental system structure and reliability problems.
- **Significant software change**: Software change can be manifested both individually and cumulatively. For example, a significant *individual* change would require changing more than one-third of the system at any one time, with the changes spread throughout the system.[55] *Cumulative* sets of changes amounting to two-thirds from significant baseline also must be managed carefully. While difficulties may not yet be apparent, such a degree of change most likely indicates a structure that is becoming increasingly frail.
- **Hardware replacement**: Hardware upgrade or replacement provides an opportunity to also upgrade the software design to take advantage of new hardware capabilities. In general, old software running on new hardware may not take advantage of improved hardware capabilities such as additional memory and graphical display capabilities. As such, maximum performance improvement will not be achieved.

8.4.4. Translating Languages

Automatic translation from one programming language to another, while an intriguing idea, should be approached cautiously.

Automatic translation (through the use of computer-based language translation tools) of existing code into Ada is sometimes proposed when an existing system is upgraded. While a number of translation systems exist, and translation offers the possibility of getting something running quickly, which may be very appealing in today's fiscal climate, there are significant problems and issues to be considered before taking this approach:

- **Degree of translation**: There are always some inherent incompatibilities between any two languages.[56] Not all constructs in other languages can be translated automatically to Ada with complete accuracy. A good translation system will flag possible mistranslations so they can be manually inspected. If the existing software contains many problematic constructs, much manual effort will be required to detect and fix possible mistranslations.
- **Loss of software engineering benefits**: The translated code will most likely be more difficult to maintain than the original software and certainly more difficult to maintain than a system designed and written in Ada from the start. Two example problem areas are:

[55] DoD policy [79] requires that Ada be used for major software upgrades of existing systems. A major software upgrade is the redesign or addition of more than one-third of the software.

[56] Some languages, notably FORTRAN, have many different dialects, which makes translating even between dialects of the same language difficult.

    - **Readability**: Automatic translators may generate strange, non-descriptive variable names that decrease understandability of the code.
    - **System structure**: Ada allows subprograms and data to be logically organized into *packages*, which improve software understandability and modifiability and allow a software developer to express intent more fully than in other languages. If such a grouping is not present in the original system, an automatic translation system *cannot* provide it.

- **Special routines**: Depending on the language being translated, there may be special functions and capabilities such as those provided by the runtime system that are not directly translatable. These special functions and capabilities may require that significant new code be written to mimic them in the new language, at unknown consequences in program schedule and performance of the resulting software.
- **Testing**: Translation does not guarantee that the resulting code will function in a manner comparable to the original code. For example, comparable language constructs may *not* be implemented with comparable efficiency. In addition, the translated code may stress the compiler in unusual ways, revealing new compiler bugs. In short, rigorous functional and performance testing are required on the translated code. If acceptance tests exist for the original system, they should be used.

In summary, language translation *must* be viewed as a risky, short-term proposition that will yield results of unknown quality. Reengineering, redesigning, and recoding the system in Ada or interfacing newly written Ada code with the current implementation are much preferred approaches.

If a translation strategy is being considered, an in-depth investigation of commercial and public domain translators is strongly suggested as part of the feasibility study. This investigation should use a robust set of test cases and benchmarks that are typical of the particular application. These benchmarks should also be used to identify the amount of rework remaining after translation to fix problems and successfully complete regression testing.

8.5. For More Information . . .

The following resource recommendations are intended as an initial starting point for those seeking more information about the topics in this chapter.

Reuse

- *Reuse: Where to Begin and Why* [122]
- *An Organized, Devoted, Project-Wide Reuse Effort* [37]
- *Software Reuse: Managerial and Technical Guidelines* [123]
- *STARS Reuse Concepts* [210]

Portability

- *Ada Quality and Style Guide* [138]
- *Portability and Style in Ada* [178]
- *The Myth of Portability in Ada* [96]

Ada Bindings

- *Available Ada Bindings* [133]

9. From Ada 83 to Ada 9X

Ada 83 is in use today, and a language revision process leading to a revised language standard (Ada 9X) is ongoing. Among the changes to be incorporated in the Ada 9X revisions of the Ada language standard are improvements in object-oriented programming, programming in the large, and real-time capabilities. Ada 9X is planned to be standardization in late 1994 with validated compilers becoming available following standardization.

9.1. Ada Language Revision Process

All standards must be periodically reaffirmed, revised, or allowed to lapse. The Ada 9X effort is making such a set of revisions to the Ada language.

Part of the development and maturation embodied in an international standard is the process by which standards organizations (such as ISO or ANSI) require that each standard be periodically reaffirmed, revised, or allowed to lapse. This is not unique to Ada; all standards (including standardized languages) must undergo this process periodically.

The Ada language standard [13] was published in 1983. In 1988, it was recommended that the Ada standard be revised. Subsequently, the Ada 9X Project Office[57] was established and, in early 1989, an Ada 9X Project Plan [11] was put into place.

DoD, NIST (because Ada is also a government standard — FIPS PUB 119), and ANSI approval of the Ada 9X standard is expected in 1994, as shown in Figure 9-1. Current plans call for DoD and NIST adoption of the approved ANSI standard for Ada 9X. Although ANSI approval of the Ada 9X standard is planned for late 1994, initial implementations may be available sooner, perhaps even before Ada 9X compiler validation is available. ISO approval of the ANSI standard may take at least two years after ANSI approval, due to voting procedures.

9.2. The Ada 9X Project

The Ada 9X Project is the effort to revise the standard for the Ada programming language, obtain adoption of the revised standard, and effect a smooth transition from Ada 83 to Ada 9X.

The goal of the Ada 9X Project has been to revise ANSI/MIL-STD-1815A to reflect current essential requirements while minimizing negative impacts and maximizing positive impacts on the Ada community. On the other hand, as the design of the Ada language is based on late 1970s programming language technologies, it is important to make enhancements to meet user needs in the many diverse applications that Ada successfully supports today.

The Ada 9X Project has tried to balance the necessary changes for Ada’s continued growth in the 1990s with the need for stability in terms of preserving the integrity of existing Ada software and tools. The project has tried in a number of ways to balance these concerns, as demonstrated in the *Ada 9X Project Plan* [11] and the *Ada 9X Transition Plan* [12].

[57]See Appendix B.3 for contact information.

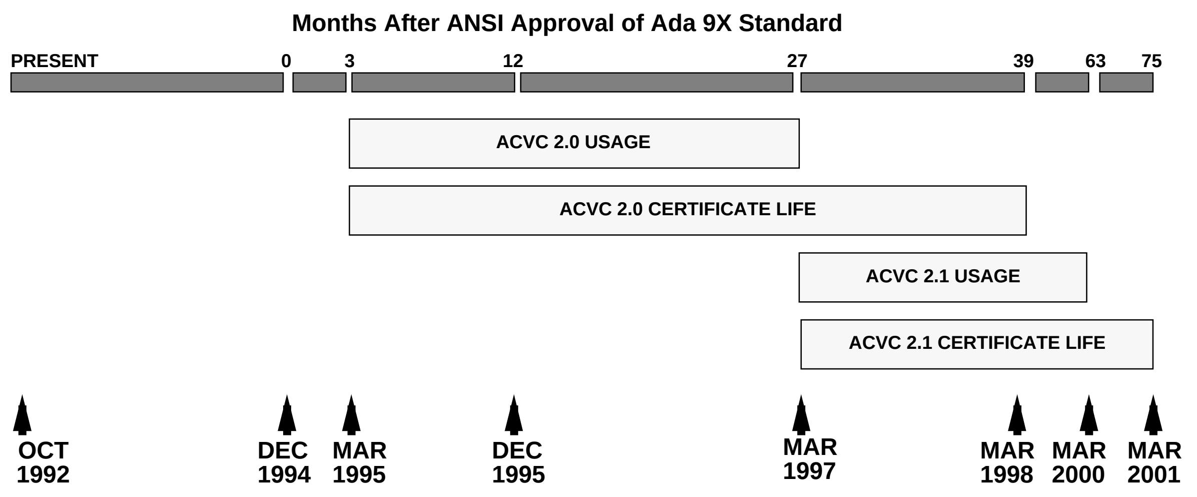

Figure 9-1: Projected Ada 9X Schedule

The Ada 9X Project Office has had wide participation, both within and outside DoD, including the:

- Requirements Team (collated more than 750 revision requests from the public and more than 800 language issues and questions since Ada 83; performed tradeoff studies of proposed changes; developed requirements for Ada 9X [71])
- Mapping/Revision Team (mapped requirements into proposed language solutions)
- Distinguished Reviewers (from industry, government, academia, tool vendors, system producers)
- ACVC Team
- ACVC Reviewers
- User/Implementer Teams (prototyped usage of proposed language changes in compilers and multiple applications areas)
- Language Precision Team (ensured that no ambiguities were being introduced by changes to the language)

The *Ada 9X Transition Plan* [12] describes activities that the Ada 9X Project will be undertaking to assist in the transition from Ada 83 to Ada 9X. Three goals have driven this transition planning:

1. Meet user needs as soon as possible.
2. Promote the development of high quality Ada 9X products.
3. Recognize that there are diverse customer bases within the Ada community and that different vendors may support one or more of these customer bases.

The effect on managers, programmers, vendors, educators, authors, and various application domains will be considered in this transition planning.

9.3. Transitioning to Ada 9X

Ada 9X will not introduce radical changes to Ada 83. Rather, enhancements are being planned to minimize incompatibilities, as Ada 9X is planned to maximize upward compatibility with Ada 83.

The revisions leading to Ada 9X are intended to include only those changes that improve the usability of the language while minimizing the disruptive effects of changing the standard. This is critical, as a significant infrastructure and investment exist in Ada 83: its programs, programmers, training material, tools, and experience base. Ada 9X is planned to be upwardly compatible with Ada 83 (i.e., most Ada 83 programs will compile and run under Ada 9X compilers).

Three major enhancements are planned in Ada 9X [215]:

1. improvements in object-oriented programming
2. programming in the large
3. real-time and parallel programming capabilities

It took several years after the release of Ada 83 to achieve production-quality compilers for numerous applications. The Ada 9X Project is taking several steps to hasten the availability of usable Ada 9X tools. Among these steps is a change in the concept of the Ada language standard in terms of a *core* language and several *annexes*, which provide extended features for specific application areas. Thus, Ada 9X compilers are anticipated to be available in several phases:

- Early-release compilers probably sometime during 1994 (before the Ada 9X standard is approved).
- Application-oriented, production-quality compilers, which implement the core language and selected annexes, within two years after the standard is approved (approximately 1996).
- Full, production-quality Ada 9X compilers, implementing the full Ada 9X standard (core and annexes), within three years after the standard is approved (approximately 1997).

However, great care is being taken to avoid sudden mandates to use Ada 9X prior to having the necessary Ada 9X infrastructure well established (e.g., tools, training). More importantly to the program manager, some Ada 9X issues must be addressed (e.g., which Ada standard to apply, when to upgrade the system to Ada 9X or to continue doing system upgrades using Ada 83 tools and language).

A subgoal of the Ada 9X efforts was to maximize upward compatibility from Ada 83 in the addition of new features and capabilities to the Ada 9X language. Complete upward compatibility is not feasible, given the scope of changes being made. However, the number of incompatibilities is small.[58] The Ada 9X Project Office will produce a document describing what incompatibilities exist and how to write Ada 83 software to minimize changes in transitioning to Ada 9X. The Ada 9X Project is also developing tools that will examine Ada 83 software to identify potential incompatibilities and recommend appropriate Ada 9X alternatives. For example, software that is written in Ada 83 can be later compiled with an Ada 9X compiler after the Ada 9X tool has been used to identify any incompatibilities. Vendors can expect a gradual phase-in and a more stable test suite concentrating on language usage. Users can also expect a gradual phase-in, with the norm being program managers making the decision on when to shift to Ada 9X.

[58]The proposed Ada 9X revision is upwardly compatible for most existing Ada 83 applications. Most incompatibilities are restricted to combinations of features that are rarely used in practice. (A thorough analysis of the upward incompatibilities is provided in [5].)

The Ada 9X Project is also taking several steps to hasten the availability of usable Ada 9X tools. For example, one such effort is the GNU Ada 9X compiler, which will be available from the Free Software Foundation (meaning that free source code for an Ada 9X compiler will be widely available).[59]

Under the proposed policies, new projects will have the choice of using Ada 83 or Ada 9X until the final Ada 9X validation suite (ACVC 2.1) is officially released (approximately 24 months after ANSI approval) [12]. The new projects, which are scheduled for completion before ACVC 2.1 is released and that have software end-products with a life expectancy of three years or more, must use Ada 83 (i.e., ACVC 2.0 will not be used for project registration[60] of compilers for long-lived projects). Issues regarding compiler performance and quality will have to be evaluated as part of the compiler evaluation and selection process.

As with today's policies, existing non-Ada projects will transition to Ada if a major project upgrade is planned. Existing Ada 83 projects can transition to Ada 9X if it is determined that Ada 9X brings necessary or desirable functionality to the project. For Ada 83 projects using a project compiler, the "major project upgrade" rule does not apply (i.e., Ada 83 projects using a validated Ada 83 *project compiler* do not have to upgrade to Ada 9X).

However, if it can be determined that a program will make future use of Ada 9X enhancements, the program may choose to design/code in Ada 83 to be as compatible as possible with Ada 9X (i.e., consider life-cycle upgrades ahead of time).[61] This might be a wise *investment* by the program office now, even though the resulting impact will not be felt until the future. This is very similar to dealing with any other preplanned product improvement (P^3I) opportunities — planning ahead for a technology upgrade during development.

The transition to Ada 9X will not be totally transparent; it will require attention to some of the transition lessons learned from Ada 83 — improving people, tools and tasks to deal with Ada 9X and its improved capabilities — and a wise program manager will anticipate this.

---

[59]The Ada 9X Project Office, in coordination with the AJPO, DARPA, and the DoD's Corporate Information Management (CIM) initiative, is sponsoring the development of GNU Ada 9X. New York University is working closely with the Free Software Foundation (FSF) to develop a GNU Ada 9X compiler written in Ada. Release is planned prior to Ada 9X approval to allow users the opportunity to have early hands-on experience with some of the Ada 9X enhancements. However, this compiler may not implement all of the proposed Ada 9X standard, including the core language and annexes. Compilers will be targeted to personal computers, the Sun SPARC workstation, and the Sun multiprocessor. Additionally, implementations of bindings to the X Window System, POSIX, and Mach are also planned.

[60]A *project compiler* is a validated Ada compiler that is selected by a specific project and baselined in accordance with applicable configuration management practices. A project compiler may be used for the life of the project. Project registration of the compiler is recommended by [3, Appendix F]. Project registration is accomplished by having the project manager inform the AJPO that the project will be using a particular compiler beyond its certificate life.

[61]A set of twelve coding guidelines for Ada 83 programmers has been developed which ensures that Ada 83 code will not encounter the few presently existing incompatibilities, once this code gets transitioned into Ada 9X environments. These guidelines are planned to be published in a paper by Dr. Erhard Ploedereder, Chairman of the Ada 9X Distinguished Reviewers, in *Ada Letters*. These guidelines are also available on the Ada 9X electronic bulletin board.

9.4. Ada 9X Validation

Ada validation policies are intended to support the transition to Ada 9X.

The Ada Compiler Validation Capability (ACVC) is the suite for testing conformance to the language standard. The two versions of the ACVC planned for Ada 9X are shown in Figure 9-1. The first version, ACVC 2.0, is planned to be officially released approximately three months after ANSI approval of Ada 9X.

All Ada validation certificates issued for validations completed with ACVC Version 1.10 expired on December 1, 1990. ACVC Version 1.11 is currently the official validation test suite for Ada 83 and will remain in effect until after ANSI adoption of Ada 9X. ACVC Version 1.11 may have test programs withdrawn but there will be no additions, modifications, or re-issuance of an ACVC version that measures conformity with the standard until Ada 9X. The certificates associated with validations completed with ACVC Version 1.11 will remain current until release of the full Ada 9X ACVC. This extended life for ACVC 1.11 means that there will be an overlap period between ACVC 1.11 (for ANSI/MIL-STD 1815A validations) and ACVC 2.0 (for Ada 9X validations) allowing users the choice of Ada 83 or Ada 9X until ACVC 2.1 (full Ada 9X testing) is required.

The first release of the ACVC for Ada 9X (ACVC 2.0) will consist of most of the Ada 9X tests and will be modularly constructed[62] so that vendors may choose the set of tests that best satisfies their customer base. The number of modules will be kept small, and vendors must pass complete modules to get credit for passing a particular validation module. All vendors must pass a core set of ACVC modules to obtain compiler validation.

The next ACVC release (ACVC 2.1) will be similar to ACVC 2.0 but will be more complete, providing coverage of the core Ada 9X languages and the annexes. This approach will aid in avoiding some of the delays that had been encountered waiting for complete, validated Ada 83 compilers and will allow quality products to reach users with the enhanced features that they need for their application domains. Thus, early vendor implementations can be tailored to user needs. For example, if a real-time oriented compiler is available and validated under ACVC 2.0, real-time users will not initially be penalized waiting for a full ACVC 2.1 compliant compiler that implements all Ada 9X features. Some compiler vendors may decide to skip ACVC 2.0 and put all their resources into validating Ada 9X compilers with the more complete ACVC 2.1.

[62]This reflects the modularity of the Ada 9X language. As with Ada 83, there is a *core* language, which must be implemented in its entirety. In Ada 9X, several annexes are defined, which provide extended features for specific application areas. These annexes provide standard definitions for application-specific requirements for: systems programming, real-time systems, distributed systems, information systems, safety and security, and numerics. The Ada 9X language will be defined in terms of the core language, which is mandatory for compiler validation, and the annexes, which are, for the most part, optional.

9.5. For More Information . . .

Recommended readings for those seeking more information about the topics addressed in this chapter include:

Ada 9X

- *Ada 9X Project Plan* [11]
- *Ada 9X Requirements* [71]
- *Ada 9X Transition Plan* [12]
- *Ada 9X Mapping*:
  - *Volume 1 — Mapping Rationale* [5]
  - *Volume 2 — Mapping Specification* [6, 7]

Appendix A: Ada Information

A.1. Readings

Recommended readings for those seeking fundamental Ada information include:

- *Ada Adoption Handbook: A Program Manager's Guide* [this volume].
- *Ada Adoption Handbook: Compiler Evaluation and Selection* [226].
- Proceedings of TRI-Ada conferences [108, 97, 38, 99].
- *Ada Language Reference Manual* [13].
- Beginning programming texts using Ada, such as [102, 199], or other texts that introduce Ada to readers with prior programming experience, such as [59], [36], [206], or [43, 171].

Collections of information about Ada include the ACM's *Resources in Ada* [111]; *Ada: Sources and Resources* [180]; and various bibliographies available through the Ada Information Clearinghouse (see below).

Ada 9X publications include:

- *Ada 9X Requirements* [71].
- *Ada 9X Transition Plan* [12].
- *Ada 9X Mapping*:
    - *Volume 1 — Mapping Rationale* [5]
    - *Volume 2 — Mapping Specification* [6, 7]

A.2. Resources

A primary source for Ada-related information is the Ada Information Clearinghouse (AdaIC). The AdaIC supports both the AJPO and the Ada 9X Project Office by distributing Ada-related information (see Section B.3 for more information).

Ada information is also available through various electronic sources. This information is often more up to date and more readily available than traditional printed materials. Various Ada-related electronic bulletin board services (BBS) are available:

- Ada Information Clearinghouse — (800) 232-9925
- Ada 9X Electronic Bulletin Board — (800) ADA-9X25

A source of much Ada information, including information from the Ada Information Clearinghouse and the Ada 9X Project Office, is an Internet-accessible computer (ajpo.sei.cmu.edu, 128.237.2.253). This machine is sponsored by the Ada Joint Program Office (AJPO) and is accessible to anyone who has ftp (file transfer protocol) access to the AJPO machine. On the AJPO machine, the directories contain Ada 9X documents and the contents of the Ada Information Clearinghouse electronic bulletin board service (BBS). For example, an up-to-date list of validated Ada compilers is available from the AdaIC through its electronic bulletin board services (BBS) or from this Internet-accessible computer.

The following sample session illustrates an Internet user accessing the AJPO machine. Those on other networks will need to consult their system managers to determine whether they have ftp access to these Internet databases.[63]

In this script, which retrieves the latest listing of validated compilers, lines on which the user enters information are preceded with asterisks. The commands and information that the user types are shown in italics. It should be noted that "cd" is the UNIX "change directory" command. At any directory, the user may type "ls" for a listing of the files or directory entries that are contained in the current directory.

To retrieve the latest validated compiler list:

1. Connect to the AJPO machine.

```
*   ftp ajpo.sei.cmu.edu
    Connected to ajpo.sei.cmu.edu.
    220 ajpo.sei.cmu.edu FTP server (SEI Version 4.3 Fri Jun 19 13:27:55 EST 1992) ready.
```

2. Login to the AJPO machine.

```
*   Name (ajpo.sei.cmu.edu:): anonymous
*   Password (ajpo.sei.cmu.edu:anonymous): your name here
    331 Guest login ok, send ident as password.
    230 Guest login ok, access restrictions apply.
```

3. Move to the public directory containing Ada information. NOTE: ftp> is the prompt from the AJPO machine.

```
*   ftp> cd public/ada-info
    250 CWD command successful.
*   ftp> ls
    200 PORT command successful.
    150 Opening data connection for /bin/ls (128.237.1.26,3917) (0 bytes).
```

At this point, the listing of all the files in the *ada-info* directory will appear. This listing is quite long, and may scroll off your screen. This example shows only the last few lines of the listing:

```
    val-comp.hlp.01Oct92
    val-doc.hlp.19May92
    val-nov.hlp.01Dec90
    val-proc.hlp.01Aug90
    valfacil.hlp.05Dec91
    vsr-docu.hlp.29Jul92
    withdrwn.hlp.08Aug91
    x-survey.hlp.01Nov91
    226 Transfer complete.
    1881 bytes received in 2.73 seconds (0.67 Kbytes/s)
```

4. Determine the correct filename to retrieve the current list of validated compilers. The filename begins with **val-comp.hlp.** and ends with the as-of date of the file. In this example, we are retrieving the file val-comp.hlp.01Oct92.

[63]The best resource on the various networks and how to navigate among them is given in *The Ada Software Repository and the Defense Data Network* [61].

5. Retrieve the current list of validated compilers, using the correct filename (including the as-of date at the end of the filename).

```
*    ftp> mget val-comp.hlp.01Oct92
*    mget val-comp.hlp.01Oct92? y
     200 PORT command okay.
     150 Opening data connection for val-comp.hlp.01Oct92 (128.237.1.26,3919)(154176 bytes).
     226 Transfer complete.
     local: val-comp.hlp.01Oct92 remote: val-comp.hlp.01Oct92
     161570 bytes received in 0.85 seconds (1.9e+02 Kbytes/s)
```

6. Logoff the AJPO machine.

```
*    ftp> quit
     quit
     221 Goodbye.
```

7. The file containing the list of validated compilers will appear in your current directory. This file is currently named valcomp.hlp.01Oct92; the date at the end of the file name will change as the file is updated by the AdaIC.

There are other sources of useful Ada information. For example, Grebyn Corporation[64] has available an annotated Ada reference manual that intersperses all the approved language commentaries in their proper place in the language reference manual.

Asset Source for Software Engineering Technology (ASSET) is a focal point for software reuse and provides reusable software components, based in the ASSET library, as well as descriptions and locations of components available from other software reuse libraries. See Section B.3 for additional information on ASSET.

[64]Contact Grebyn Corporation, P.O. Box 497, Vienna, VA 22183-0497, (703) 281-2194, E-mail: products@grebyn.com, for further information.

Appendix B: Ada Working Groups and Resources

Many groups are involved in Ada technology activities. In the following sections, details of various groups are discussed, including:

- professional organizations
- standards organizations
- U.S. government-sponsored/endorsed organizations
- DoD Software Executive Officials

Contact information is provided; where known, Defense Switched Network (DSN, formerly AUTOVON numbers) and/or electronic mail (E-mail) addresses are also included.

B.1. Professional Organizations

B.1.1. Ada Joint Users Group (AdaJUG)

Ada Joint (Services) Users Group (AdaJUG) (formerly Ada-JOVIAL Users Group) is a non-profit organization whose purpose is to encourage a dialogue between Government and industry concerning Ada issues. The AdaJUG makes recommendations to appropriate military services and DoD agencies regarding language policies and practices. The current AdaJUG chair is:

Mike Ryer
Intermetrics
733 Concord Ave.
Cambridge, MA 02138
(617) 661-1840

B.1.2. SIGAda

The Association for Computing Machinery Special Interest Group on Ada is a scientific association focused on the Ada language. *Ada Letters* is the SIGAda bimonthly publication. The current SIGAda chair is:

Mark Gerhardt
ESL Inc., MS M507
495 Java Drive
Sunnyvale, CA 94088-3510
(408) 752-2459 or (408) 738-2888 (Switchboard)
Email: gerhardt@ajpo.sei.cmu.edu

SIGAda sponsors a number of working groups. These and current points of contact are normally listed in *Ada Letters*.[65] The following are the current SIGAda working groups:

- **AIWG**: Artificial Intelligence Working Group
- **ARTEWG**: Ada Runtime Environment Working Group
- **CAISWG**: CAIS Working Group
- **CAUWG**: Commercial Ada Users Working Group

[65]To obtain information about membership or *Ada Letters*, contact ACM, 1515 Broadway, New York, NY 10036-9998, (212) 869-7440, or e-mail acmhelp@acmvm.bitnet. Single copies of *Ada Letters* may be purchased from the ACM Order Dept. at (800) 342-6626 or (410) 528-4261.

- **EDWG**: Educational Products Working Group
- **NUMWG**: Numerics Working Group
- **OBJWG**: Object-Oriented Working Group
- **PIWG**: Performance Issues Working Group
- **REUSEWG**: Reuse Working Group
- **SDSAWG**: Software Development Standards and Ada Working Group
- **SSTDWG**: Secondary Standards Working Group

B.1.3. Other Professional Organizations (World-Wide)

While SIGAda and AdaJUG are the predominant U.S. professional Ada organizations, similar organizations exist in Europe and around the world. In addition to Ada-Europe, there are also Ada groups associated with the Commission of the European Communities and the European Space Agency. Ada organizations also exist in Australia, Canada, Denmark, France, Germany, Greece, Ireland, the Netherlands, Norway, Scotland, Spain, Sweden, Switzerland, and the United Kingdom. Contact information for these organizations can be found in *Ada Letters*.

B.2. Standards Organizations

ISO/IEC JTC1/SC22 WG9 Ada

- **Purpose**: Standardization of the Ada programming language is the responsibility of a working group under the International Organization for Standardization and the International Electro-Technical Commission known as ISO/IEC JTC1/SC22 WG9. Ada was initially developed as a military standard (MIL-STD-1815A) and became an American National Standards Institute (ANSI) standard (ANSI/MIL-STD-1815A, 1983) through the canvass process. It is also a Federal Information Processing Standard (FIPS 119, 1985) of the National Institute of Standards and Technology (NIST) as an endorsement of the Ada standard as created by others. All the versions of the standard are identical to ISO 8652:1987.

  Membership in WG9 is held by national standards organizations rather than individuals. About twenty countries are actively participating in the work. In the case of the United States, the national body is ANSI, which delegates the work to a group called the U.S. TAG (Technical Advisory Group) for WG9. The TAG is responsible for developing national positions and naming members of the delegations to attend WG9 meetings. The TAG is co-chaired by Christine Anderson (program manager for Ada9X) and John Solomond (director of the Ada Joint Program Office).

  WG9 meets at least twice yearly and is led by its convener, Robert Mathis (USA), a former director of the Ada Joint Program Office. Most of the work of WG9 in drafting standards is performed in smaller bodies called "rapporteur groups." In principle, every proposal for a standard is formulated as a "work item." New work items are approved at the JTC1 level and eventually assigned to a rapporteur group. The standards drafted by the rapporteur groups work their way up through the ISO hierarchical balloting process with successive designations as Committee Draft (CD), Draft International Standard (DIS) and, finally, International Standard (IS).

  Currently there are eight rapporteur groups within WG9. Although the work of these groups is generally focused on the current Ada standard, all of them are also actively participating in the development of the proposal for Ada 9X:

  1. Ada Rapporteur Group (ARG)
  2. Character Rapporteur Group (CRG)
  3. Information Systems Rapporteur Group (IRG)

4. Numerics Rapporteur Group (NRG)
5. Real-time Rapporteur Group (RRG)
6. SQL Rapporteur Group (SRG)
7. Uniformity Rapporteur Group (URG)
8. Ada 9X Rapporteur Group (XRG)

Not all standardization related to Ada occurs in WG9. Sometimes bindings from standards to particular languages are performed by the language committee, and sometimes they are performed by the committee that created the standard. For example, the Ada binding to POSIX is under ISO/IEC JTC1/SC22 WG15 and IEEE P1003.5.

For further information, contact:

Dr. Robert Mathis, Convener
ISO/IEC JTC1/SC22 WG9 Ada
Suite 1815
4719 Reed Road
Columbus, Ohio 43220
(614) 538-9232
Email: mathis@ajpo.cmu.sei.edu

POSIX /Ada working group:

- **Purpose**: Develop an Ada binding[66] to the application programming interfaces (API) defined in the POSIX operating system specification ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990).

IEEE P1003.5 is a single committee that has been working on two projects:

1. **Ada binding to POSIX:** The P1003.5 project objective has been to provide Ada Language interfaces to the P1003.1 POSIX operating system specification. This work has resulted in the adoption of IEEE Std 1003.5-1992, which defines a binding to IEEE Std 1003.1.
2. **Ada binding to the IEEE Std 1003.4 POSIX real-time extensions:** The P1003.20 project objective is to define an Ada language binding to the P1003.4/4a real-time POSIX extensions.

For further information, contact:

James P. Longers
Paramax Systems Corporation
70 E. Swedesford Rd.
P.O. Box 517
Paoli, PA 19301 U.S.A.
email: longers@prc.unisys.com
(215) 648-2670

In addition, an electronic mailing list has been established for those interested in POSIX-Ada binding. For more information, or to subscribe to the list, send computer mail to posix-ada-request@verdix.com.

[66]The POSIX Ada Language Interface standard is contained in IEEE Std P1003.5.

B.3. U.S. Government Sponsored/Endorsed Organizations

AJPO: Ada Joint Program Office

- **Purpose**: Oversees the total direction of the Ada program. The AJPO reports to the Deputy Undersecretary of Defense for Research and Advanced Technology (DUSDR&AT). For further information, contact:

  Dr. John Solomond, Director
  Ada Joint Program Office
  The Pentagon, Room 3E118
  Washington, D.C. 20301-3081
  (703) 614-0208
  Fax: (703) 685-7019
  Email: solomond@ajpo.sei.cmu.edu

Ada 9X Project Office:

- **Purpose**: Responsible for revising the Ada 83 standard. (See Chapter 9 for more information on the Ada 9X Project.) For further information, contact:

  Christine Anderson
  Ada 9X Project Office
  PL/VTET
  Kirtland AFB NM 87117-6008
  (505) 846-0817
  Fax: (505) 846-2290 - Attn: Chris Anderson
  Email: andersonc@plk.af.mil

Ada Federal Advisory Board:

- **Purpose**: A federal advisory committee, composed of compiler developers, language designers, embedded system users, educators, and government personnel that provides the Director of the AJPO "with a balanced source of advice and information regarding the technical and policy aspects of the Ada Program [4]." For further information, contact:

  Dr. John Solomond, Director
  Ada Joint Program Office
  The Pentagon, Room 3E118
  Washington, D.C. 20301-3081
  (703) 614-0208
  Fax: (703) 685-7019
  Email: solomond@ajpo.sei.cmu.edu

AdaIC: Ada Information Clearinghouse

- **Purpose**: Supports the AJPO by distributing Ada-related information, including:
  - policy statements
  - lists of validated compilers
  - classes
  - conferences
  - text books and serials relating to Ada
  - projects using Ada

  Also supports the Ada 9X Project Office by serving as the distribution point for information related to the Ada 9X Project.[67]

[67]The Ada 9X documents are also available electronically on the Ada 9X BBS at (800) Ada-9X25 or (301) 459-8939.

In addition to publishing a free quarterly newsletter, an electronic bulletin board system (300 through 9600 baud, no parity, 8 bits, 1 stop bit) is available at (703) 614-0215. This bulletin board also provides access to the AdaIC's Ada products and tools, Ada article abstracts, and Ada bibliography databases.

The Ada Information Clearinghouse (AdaIC) also maintains files on the AJPO host, which are available via anonymous FTP from the public directories on the AJPO host computer (ajpo.sei.cmu.edu).

For further information, contact:

Ada Information Clearinghouse
c/o IIT Research Institute (IITRI)
4600 Forbes Blvd.
Lanham MD 20706-4312
(800) Ada-IC11 or (703) 685-1477
Fax: (703) 685-7019
Email: adainfo@ajpo.sei.cmu.edu; CompuServe 70312,3303

IIT Research Institute operates the AdaIC for the Ada Joint Program Office.

AVO: Ada Validation Organization

- **Purpose**: The AVO is directly responsible to the AJPO. The AVO coordinates and administers compiler validation policy and procedures. For further information, contact:

Ada Validation Organization
Institute for Defense Analyses
ATTN: Audrey Hook
1801 Beauregard Street
Alexandria, Virginia 22311
(703) 845-6639
FAX: (703) 845-6848
Email: hook@ida.org

ACVC Review Team: Ada Compiler Validation Capability Review Team

- **Purpose**: The ACVC Review Team provides expert technical review for the Ada Compiler Validation Capability. For further information, contact:

Dr. Nelson Weiderman
127 Schooner Drive
Wakefield, RI 02879
(401) 783-6863
Email: nhw@sei.cmu.edu

AMO: ACVC Maintenance Organization

- **Purpose**: Responsible for the development, maintenance, and support of the Ada Validation Suite (AVS), which includes the ACVC and the ACEC. Additionally, the AMO supports Ada language maintenance activities. For further information, contact:

  Steve Wilson
  ASC/SCEL
  Wright-Patterson AFB, Ohio 45433-6503
  (513) 255-4472
  Fax: (513) 255-4585
  Email: wilsons@adawc.wpafb.af.mil

  Ada-JOVIAL Newsletter: The Standard Languages and Environments Division, which operates the AMO and the Department of Defense Ada Validation Facility at Wright Patterson AFB, Ohio, also publishes the *High Order Language Control Facility Ada-JOVIAL Newsletter.* For further information on the *Ada-JOVIAL Newsletter*, contact:

  Dale Lange
  Standard Languages and Environments Division
  ASC/SCEL
  Wright-Patterson AFB, OH 45433-6503
  Tel: (513) 255-4472
  E-mail: langed@adawc.wpafb.af.mil

ASEET: Ada Software Engineering Education and Training Team

- **Purpose**: The Ada Software Engineering Education and Training (ASEET) Team is composed of representatives from the Army, Air Force, Navy, Marine Corps, other DoD agencies, and academia. The team conducts workshops and symposia for Ada educators within DoD and academia and coordinates the activities of DoD organizations engaged in meeting the Ada education and training needs.

  An ASEET resource library of educational materials, the ASEET Material Library (AML), is located at the Ada Joint Program Office. This material is not copyrighted and can be obtained by DoD personnel and copied at cost by non-DoD personnel.

  For further information, contact:

  Capt. David A. Cook
  Dept. of Computer Science, DFCS
  US Air Force Academy, CO 80840
  (719) 472-3590
  DSN 259-3131
  E-mail: dcook@ajpo.sei.cmu.edu

  Ada Software Engineering Education and Training Team
  Institute for Defense Analyses
  1801 North Beauregard Street
  Alexandria, VA 22311
  Attn: Resource Staff Member
  (703) 845-6626

AVF: Ada Validation Facility

- **Purpose**: Responsible for validation of Ada compilers (giving priority to DoD targeted compilers) and registration of derived compilers.[68] Ada Validation Facilities currently exist in the US (2), the UK, France, and West Germany. For further information, contact one of the Ada Validation Facility Managers:

  Captain Russ Hilmandolar
  Ada Validation Facility
  Language Control Facility ASC/SCEL
  Bldg. 676 / Rm. 135
  Wright-Patterson AFB, OH 45433-6503
  Tel: (513) 255-4472
  Fax: (513) 255-4585
  E-mail: hilmanrk@adawc.wpafb.af.mil

  Fabrice Garnier de Labareyre or Alphonse Philippe
  AFNOR
  Tour Europe, Cedex 7
  F-92049 Paris La Defence
  FRANCE
  Tel: +33-1-42-91-5960
  Telefax: +33-1-42-91-5911

  Jon Leigh or Dave Bamber
  National Computing Centre Limited
  Oxford Road
  Manchester M1 7ED
  ENGLAND
  Tel: +44 (61) 228 6333
  FAX: +44 (61) 228-2579

  Dr. William Dashiell
  National Institute for Standards and Technology
  National Computer Systems Laboratory
  Bldg. 255/Room A266
  Gaithersburg, MD 20899
  Tel: (301) 975-2490
  Fax: (301) 590-0932

  Michael Tonndorf
  IABG, Dept ITE
  Einsteinstrasse 20
  W-8021 Ottobrunn
  GERMANY
  Tel: +49-89-6088-2477

[68]Derived compilers are defined on page 54.

For further information on the AVFs or validation policies and procedures, contact:

Ada Validation Organization
Institute for Defense Analyses
Attn: Audrey Hook
1801 North Beauregard Street
Alexandria, VA 22311
Tel: (703) 845-6639
FAX: (703) 845-6848
Email: hook@ida.org

Software Technology for Adaptable, Reliable Systems (STARS)

- **Purpose:** STARS is a technology development, integration, and transition program to demonstrate a process-driven, domain-specific, reuse-based approach to software engineering (also known as *megaprogramming*) that is supported by appropriate tools and software engineering environment (SEE) technology.

  The goal of the STARS Program is to increase software productivity, reliability, and quality by synergistically integrating support for modern software development processes and modern reuse concepts into state-of-the-art software engineering environment (SEE) technology. To meet that goal, STARS has set a number of objectives for the 1992-94 time frame:

  - **Software Reuse:** Establish the basis for a paradigm shift to reuse-based software development.
  - **Software Process:** Establish capabilities for process definition and management, and for tailoring processes to the needs of the application.
  - **Environments:** Establish adaptable, commercially viable SEE solutions that are built upon open architecture industry standards, leverage commercial products, and include automated support for process management and software reuse.
  - **Transition/Demonstration/Validation:** Demonstrate that the STARS integrated reuse, process, and SEE solutions can be used in actual practice to increase the quality and life-cycle supportability of DoD software products.

  STARS maintains a multi-level technology transition affiliates program to provide an opportunity for the DOD software community (including tool vendors) to participate in STARS technical activities.

  For further information, contact:

  John Foreman
  Software Technology for Adaptable, Reliable Systems
  801 N. Randolph Street
  Suite 400
  Arlington, VA 22203
  (703) 243-8655
  E-mail: affiliates-desk@stars.rosslyn.unisys.com

Asset Source for Software Engineering Technology (ASSET)

- **Purpose:** ASSET was established by the Defense Advanced Research Projects Agency (DARPA) under its Software Technology for Adaptable, Reliable Systems (STARS) Program.[69] ASSET, located in Morgantown, West Virginia, is chartered by DARPA to serve as a focal point

[69]See page 102.

for software reuse and to facilitate a national software reuse infrastructure and industry. ASSET provides its client base with reusable software components based in the ASSET library as well as descriptions and location of components available from other software reuse libraries.

The ASSET library consists of a repository containing cataloged assets, and a bibliography containing abstracts and other information needed to locate the assets contained in the repository. The repository currently contains three collections:

1. the STARS Foundation collection;
2. the STARS Catalog Products collection, which is a set of reusable software assets produced by the STARS prime contractors; and
3. newly developed STARS key assets.

In addition to cataloged items in the library, ASSET also has available other information specific to Ada, e.g., *Ada Language Reference Manual*, Ada Software Repository assets, and ANNA, a Language Extension of Ada, as well as other reusable software assets.

In addition to the library, ASSET provides other services, including the STARS bulletin board and newsgroups with STARS-specific and USENET postings. The bulletin board may be accessed by telnet(ing) to source.asset.com and entering STARSBBS at the login prompt. No account is needed.

ASSET is currently developing a National Software Reuse Directory where producers, vendors, and other reuse libraries can list selected reusable components, and where users looking for sources of reusable components will be able to view what is available, and where. This service should be available by the end of 1992.

ASSET is available to all software developers. This includes commercial, educational, and private, as well as government and government contractors. ASSET is available via internet and 9600 baud dial up modems.

For more information about ASSET and its services, or to obtain a user account, contact:

ASSET
2611 Cranberry Square
Morgantown, WV 26505
(304) 594-3954
Email: info@source.asset.com

Software Technology Support Center (STSC)

- **Purpose:** The STSC was established by Headquarters, USAF, to serve as the focal point for the proactive management of computer systems support tools, methods, environments, and Ada issues for Joint Services software activities. It is an office of the Ogden Air Logistics Center (Air Force Materiel Command), and is located at Hill Air Force Base, Utah. In addition to publishing a free newsletter, *CrossTalk*, an electronic bulletin board system is available at (801) 777-7553 (DSN 458-7553). For further information, contact:

Software Technology Support Center
Ogden ALC/TISE
Building 100, Bay G
Hill AFB, UT 84056
(801) 777-8045/7703 (DSN 458-8045/7703)
Fax: (801) 777-8069

B.4. DoD Software Executive Officials

DoD Instruction 5000.2, "Defense Acquisition Management Policies and Procedures," Part 6, Section D [79] gives the following guidance:

> **Software Executive Official.** The DoD Component Acquisition Executive will designate a senior level Software Executive official who will monitor, support, and be a focal point for Ada usage and sound software engineering, development, and life-cycle support policy and practice.

This instruction replaced DoD Directive 3405.2, "Use of Ada in Weapon Systems" [78], which had established the Ada Executive Officials (AEOs). In most cases, the individuals who served as AEOs are now serving as Software Executive Officials (SEOs).

A listing of current DoD Software Executive Officials (SEOs) (as of June 30, 1992) is shown below. A current listing is available through the AdaIC.

Service/Agency	Contact
OSD	Deputy Director, Defense Research and Engineering (S & T)
OSD	Director, Ada Joint Program Office
U.S. Army	Vice Director for Information Management, Office of the Director of Information Systems for Command, Control, Communications and Computers (DISC4/SAIS-2C)
U.S. Navy	Commander, Naval Information Systems Management Center
U.S. Air Force	Deputy Assistant Secretary of the Air Force (Communication, Computers & Logistics)
Joint Chiefs of Staff	Director, C4S
Defense Information Systems Agency (DISA)	Director, Center for Standards
Defense Security Assistance Agency	Deputy Chief, Weapons Systems Division (Plans)
Strategic Defense Initiative Organization (SDIO)	SDIO/SD
Defense Advanced Research Projects Agency (DARPA)	Director, Software and Intelligent Systems Technology Office
Defense Intelligence Agency (DIA)	Chief, DS Systems Integration (DS-SIM)
Defense Logistics Agency (DLA)	Chief, System Integration Division (Attn: DLAZI)
National Security Agency (NSA)	Deputy Director for Telecommunications and Computer Services (Technology)
C^3I	Deputy Director of Information Technology

References

1. A'Hearn, F. W., Bergmen, C. E., & Hirsch, E. *Evolutionary Acquisition: An Alternative Strategy for Acquiring Command and Control (C2) Systems (AD-A190509).* Defense Systems Management College, Fort Belvoir, VA, 1987.

2. Abbott, R. J. *Recommended Tailorings of DOD-STD-2167A and DI-MCCR-80012A for Object-Oriented Software Development.* Aerospace Corp., Los Angeles, CA, December 1991.

3. Ada Joint Program Office. *Ada Compiler Validation Procedures, Version 3.1.* Ada Joint Program Office, Office of the Secretary of Defense, Washington, D.C., 1992.

4. Ada Joint Program Office. Charter - Ada Board. Attachment to letter to Senator Barry M. Goldwater, October 2, 1986.

5. Ada 9X Mapping/Revision Team. *Ada 9X Mapping, Volume I: Mapping Rationale, Version 4.1*. Tech. Rpt. IR-MA-1249-2, Intermetrics, Inc., Cambridge, MA, March 1992.

6. Ada 9X Mapping/Revision Team. *Ada 9X Mapping, Volume II: Mapping Specification, Version 4.0*. Tech. Rpt. IR-MA-1250-2, Intermetrics, Inc., Cambridge, MA, December 1991.

7. Ada 9X Mapping/Revision Team. *Ada 9X Mapping, Volume II: Mapping Specification and Rationale (Annexes), Abridged, Version 4.1*. Tech. Rpt. IR-MA-1250-3, Intermetrics, Inc., Cambridge, MA, March 1992.

8. Ada and Software Management Assessment Working Group. *Ada and Software Management in NASA: Assessment and Recommendations.* NASA Goddard Space Flight Center, Greenbelt, MD, 1989.

9. Albrecht, A. J. and Gaffney, J. E., Jr. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". *IEEE Transactions on Software Engineering SE-9*, 6 (Nov 1983), 639-648.

10. Alvarez, A., et al. "Proceedings of the Second International Workshop on Real-Time Ada Issues". *ACM SIGAda Ada Letters (Special Edition) VIII*, 7 (1988).

11. Anderson, C. M. *Ada 9X Project Plan.* Ada 9X Project Office, Air Force Armament Laboratory, Eglin Air Force Base, Florida, 1990.

12. Anderson, C. M. *Ada 9X Transition Plan.* Ada 9X Project Office, Air Force Armament Laboratory, Eglin Air Force Base, Florida, May 1992.

13. ANSI. *ANSI/MIL-STD-1815A, Reference Manual for the Ada Programming Language.* American National Standards Institute, January 1983.

14. ANSI. *American National Standard for Information Systems - Database Language - SQL X3.125-1986.* American National Standards Institute, 1986.

15. Armitage, J.W. and Chelini, J.V. "Ada Software on Distributed Targets: A Survey of Approaches". *Ada Letters 4*, 4 (Jan/Feb 1985), 32-37.

16. Ada Runtime Environment Working Group. Catalogue of Runtime Implementation Dependencies. ACM, New York.

17. Ada Runtime Environment Working Group. A Framework for Describing Ada Runtime Environments (formerly A Canonical Model and Taxonomy of Ada Runtime Environments). ACM, New York.

18. Ada Runtime Environment Working Group. "Catalogue of Interface Features and Options for the Ada Runtime Environment (CIFO 3.0)". *Ada Letters 11*, 8 (Fall 1991).

19. Atkinson, C., Moreton, T. & Natali, A. *Ada for Distributed Systems.* Cambridge University Press, Cambridge, UK, 1987.

20. Atkinson, C. *Object-Oriented Reuse, Concurrency, and Distribution: An Ada-Based Approach.* ACM Press, New York, 1991.

21. Ausnit, C.N., Braun, C., Eanes, S., Goodenough, J. & Simpson, R. *Ada Reusability Guidelines*. Tech. Rept. ESD-TR-85-142, Softech, Waltham, MA, April 1985. Prepared for Electronic Systems Division (AFSC) under contract #F33600-84-D-0280.

22. Ausnit, C.N., Ansarov, E.R., Cohen, N.H., & Ziemba, M.V. *Program Office Guide to Ada, Edition 2*. Tech. Rept. ESD TR-86-282 (II), Softech, Waltham, MA, October 1986. Prepared for Electronic Systems Division under contract #F33600-84-D-0280.

23. Ausnit, C.N., Guerrieri, E., Ingwersen, N., & Ruegsegger, S. *Program Office Guide to Ada, Edition 3*. Tech. Rept. ESD-TR-88-102, Softech, Waltham, MA, December 1987. Prepared for Electronic Systems Division under contract #F33600-87-D-0337.

24. Ausnit, C.N., Guerrieri, E., Hood, P. & Ingwersen, N. *Program Office Guide to Ada, Edition 4*. Tech. Rept. ESD-TR-88-264, Softech, Waltham, MA, March 1988. Prepared for Electronic Systems Division under contract #F33600-87-D-0337.

25. Baker, T., et al. "Proceedings of the Third International Workshop on Real-Time Ada Issues". *ACM SIGAda Ada Letters (Special Edition) X*, 4 (1990).

26. Baker, T., et al. "Proceedings of the Fourth International Workshop on Real-Time Ada Issues". *ACM SIGAda Ada Letters (Special Edition) X*, 9 (1990).

27. Bamberger, J. "What Every Good Manager Should Know About Ada". *IEEE Aerospace and Electronics Systems Magazine 3*, 5 (May 1988), 2-8.

28. Barnes, J., et al. "International Workshop on Real-Time Ada Issues". *ACM SIGAda Ada Letters (Special Edition) VII*, 6 (1987).

29. Baskette, J. "Life Cycle Analysis of an Ada Project". *IEEE Software* (January 1987), 40-47.

30. Baumert, J. H. and McWhinney, M. S. *Software Measures and the Capability Maturity Model.* Technical Report CMU/SEI-92-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1992.

31. Baumgarten, U. Distributed Systems and Ada-Current Projects and Approaches Comparative Study's Results. In Christodoulakis, D., Ed., *Ada: The Choice for '92 (Ada-Europe International Conference Proceedings; Athens, Greece; 13-17 May 1991)*, Springer-Verlag, Berlin, Germany, 1991, pp. 260-278.

32. Bayer, J. & Melone, N. *Adoption of Software Engineering Innovations in Organizations*. Technical Report CMU/SEI-89-TR-17, ADA211573, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 1989.

33. Bishop, J. *Distributed Ada: Developments and Experiences (Proceedings of the Distributed Ada '89 Symposium).* Cambridge University Press, Cambridge, UK, 1990.

34. Boehm, B.W. Ada COCOMO: TRW IOC Version. In *Proceedings of the Third COCOMO User's Group Meeting*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

35. Boehm, B. W. "A Spiral Model of Software Development and Enhancement". *IEEE Computer* (May 1988), 61-72.

36. Booch, G. *Software Engineering with Ada. (2nd edition).* Benjamin-Cummings, 1987.

37. Bowen, G. M. "An Organized, Devoted, Project-Wide Reuse Effort". *Ada Letters 12*, 1 (Jan/Feb 1992), 43-52.

38. Brandon, C. S., & Eustice, A. S. (Ed.) *Proceedings TRI-Ada'91.* ACM, New York, 1991.

39. Brenner, N. J. *Software Development Effort: Ada vs. Other Higher Order Languages*. Technical Report TR-0017/1, Tecolote Research, Inc., Santa Barbara, CA, September 1991.

40. Brotherton, T. W. "Capability Evaluation Adapted to Procurement". *IEEE Software 9*, 3 (May 1992), 109-110.

41. Brownsword, L. Practical Methods for Introducing Software Engineering and Ada into an Actual Project. In Heilbrunner, S., Ed., *Ada in Industry: Proceedings of the Ada-Europe International Conference, Munich, 7-9 June 1998*, Cambridge University Press, Cambridge, 1988, pp. 132-140.

42. Brownsword, L. and McUmber, R. Applying the Iterative Development Process to Large 2167A Ada Projects. In Brandon, Carl S., & Eustice, Ann S., Ed., *Proceedings TRI-Ada'91*, ACM, New York, 1991, pp. 378-386.

43. Bryan, D.L., & Mendal, G.O. *Exploring Ada, Volume 1.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

44. Buckley, F. J. *Implementing Software Engineering Practices.* Wiley-Interscience, New York, NY, 1989.

45. Buxton, J. N., & Malcolm, R. "Software Technology Transfer". *Software Engineering Journal 6*, 1 (January 1991), 17-23.

46. Byrne, E. J. "Software Reverse Engineering: A Case Study". *Software - Practice and Experience 21*, 12 (December 1991), 1349-1364.

47. Byrnes, P. Software Capability Evaluation (SCE) Tutorial. Presented at SEI Software Engineering Symposium '92, Pittsburgh, PA, September 1992.

48. Carleton, A. D., Park, R. E., Goethert, W. B., Florac, W. A., Bailey, E. K., and Pfleeger, S. L. *Software Measurement for DoD Systems: Recommendations for Initial Core Measures*. Technical Report CMU/SEI-92-TR-19, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1992.

49. Carlson, M. & Smith, G.N. *Understanding the Adoption of Ada: Results of an Industry Field Survey*. Technical Report CMU/SEI-90-SR-10, ADA226725, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1990.

50. Carstensen, H. B. Experiences in Delivering a Large Ada Project. In *Proceedings of MCC'87 - Military Computing Conference*, EW Communications, Palo Alto, 1987, pp. 115-126.

51. Castor, V.L. *Issues to Be Considered in the Evaluation of Technical Proposals from the Ada Language Perspective*. Technical Report AFWAL-TR-85-1100, Avionics Laboratory (AFWAL/AAAF), Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, OH, June 1985.

52. Chastek, G.J., Graham, M.H., Zelesnik, G. *The SQL Ada Module Description Language - SAMeDL*. Tech. Rept. CMU/SEI-90-TR-26, ADA235781, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

53. Cherry, G.W. *The Pamela Designer's Handbook.* The Analytic Science Corporation, Reading, MA, 1986.

54. Cherry, G.W. . *Introduction to PAL and PAMELA II (Process Abstraction Language and Process Abstraction Method for Embedded Large Applications).* ThoughtTools, Reston, VA, 1987.

55. Cherry, G.W. *Software Construction with Object-Oriented Pictures : Specifying Reactive and Interactive Systems.* Thought Tools, Canandaigua, NY, 1990.

56. Clark, P. G. & Crawford, B. S. *Evaluation and Validation Guidebook: Version 3.0, AD-A236 494.* Avionics Directorate, Wright Laboratory, Wright-Patterson AFB, OH, 1991.

57. Clark, P. G. & Crawford, B. S. *Evaluation and Validation Reference Manual: Version 3.0, AD-A236 697.* Avionics Directorate, Wright Laboratory, Wright-Patterson AFB, OH, 1991.

58. Cochran, M. and Gomaa, H. Validating the ADARTS Software Design Method for Real-Time Systems. In Brandon, Carl S., & Eustice, Ann S., Ed., *Proceedings TRI-Ada'91*, ACM, New York, 1991, pp. 33-44.

59. Cohen, N. H. *Ada as a Second Langauge.* McGraw-Hill, New York, 1986.

60. Collingbourne, L., Cholerton, A., & Bolderston, T. Ada for Tightly Coupled Systems. In Bishop, J., Ed., *Distributed Ada: Developments and Experiences (Proceedings of the Distributed Ada '89 Symposium; Southampton, UK; 11-12 Dec. 1989)*, Cambridge University Press, Cambridge, UK, 1990, pp. 177-199.

61. Conn, R. L. *The Ada Software Repository and the Defense Data Network: A Resource Handbook.* Zoetrope, New York, 1987.

62. Connors, D. Selecting An Ada Contractor - One Way That Worked. In Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 530-542.

63. Cornhill, D.T. Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets. In *Proceedings 1984 IEEE Conference on Ada Applications and Environments*, IEEE, 1984, pp. 153-162.

64. Crafts, R. E. MIS in Ada: A Ten-Year Track Record of Success. In *Ada's Success in MIS: A Formula for Progress*, George Mason University Center of Excellence in C^{3}I, Fairfax, VA, 1992.

65. Crawford, B.S.; Jazwinski, A.H. The AdaGRAPH Tool for Enhanced Ada Productivity. In *Proceedings of the IEEE 1986 National Aerospace and Electronics Conference, NAECON'86*, IEEE, New York, NY, 1986, pp. 664-70, Vol. 3.

66. Crosier, T. *A Guide For Implementing Total Quality Management.* Tech. Rept. SOAR-7, Reliability Analysis Center, Rome, NY, 1990.

67. Cross, J.K., Kamrad, M.J., & Fernandez, S.J. Communications Among Distributed Ada Programs. In *Proceedings of the IEEE 1991 National Aerospace and Electronics Conference (NAECON 1991)*, IEEE, New York, 1991, pp. 627-632, Vol. 2.

68. Defense Science Board Task Force on Military Software. *Report of the Defense Science Board Task Force on Military Software.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1987.

69. de Gyurky, S. M. *The Global Decision Support System (GDSS) Software Methodology: The Comparison of A Non-Standard Software Approach To DOD-STD-2167A.* Report JPL D-3216, Jet Propulsion Laboratory, Pasadena, CA, March 1989.

70. Department of the Air Force, Office of the Deputy Assistant Secretary (Communications, Computers, & Logistics). Interpretation of FY 1991 DoD Appropriations Act. Memorandum, April, 1991.

71. Department of Defense. *Ada 9X Requirements.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, December 1990.

72. Department of Defense. *DOD-STD-1703(NS): Software Product Standards.* Department of Defense, Washington, D.C., April 1987.

73. Department of Defense. *MIL-STD-1750A Airborne Computer Instruction Set Architecture (includes Notice 1).* Department of Defense, Washington, D.C., May 1982.

74. Department of Defense. *DOD-STD-2167: Defense System Software Development.* Department of Defense, Washington, D.C., June 1985.

75. Department of Defense. *DOD-STD-2167A: Defense System Software Development.* Department of Defense, Washington, D.C., February 1988.

76. Department of Defense. *MIL-HDBK-287 (Military Handbook): A Tailoring Guide for DOD-STD-2167A, Defense System Software Development.* Department of Defense, Washington, D.C., August 1989.

77. Department of Defense. *DoD Directive 3405.1, Computer Programming Language Policy.* Department of Defense, Washington, D.C., April 1987.

78. Department of Defense. *DoD Directive 3405.2, Use of Ada in Weapon Systems.* Department of Defense, Washington, D.C., March 1987. Cancelled by DoD Directive 5000.1, Defense Acquisition, 23 February 1991.

79. Department of Defense. *DoD Instruction 5000.2, Defense Acquisition Management Policies and Procedures.* Department of Defense, Washington, D.C., February 23, 1991.

80. Department of the Air Force, Headquarters, Air Force Systems Command. Software Development Capability Assessment (SDCA). AFSC Pamphlet 800-51.

81. Department of the Air Force, Deputy Assistant Secretary of the Air Force (Communications, Computers, and Logistics). *Ada and C++: A Business Case Analysis.* Department of the Air Force, Washington, D.C., 1991.

82. Department of the Air Force, Headquarters, Aeronautical Systems Division (AFSC). Software Development Capability/Capacity Review. ASD Pamphlet 800-5, September, 1987.

83. Department of the Army (SAIS-ADO). HQDA LTR 25-90-1, Subject: Implementation of the Ada Programming Language. Letter, July 1990.

84. Department of the Navy, Office of the Assistant Secretary (Research, Development and Acquisition). Interim Department of the Navy Policy on Ada. Letter, June 1991.

85. Department of the Navy, Headquarters, U.S. Marine Corps (CCP-50). Marine Corps Ada Implementation Plan. Letter, March 1988.

86. U.S. Department of Transportation, Federal Aviation Adminstration, Associate Administrator for NAS Development, AND-1 and Associate Administrator for Airway Facilities, AAF-1. ACTION NOTICE - National Airspace System (NAS) Software Procedures. October 1989.

87. Director of Defense Information (OASD). *Ada's Success in MIS: A Formula for Progress.* George Mason University Center of Excellence in C^{3}I, Fairfax, VA, 1992.

88. Dobbing, B.J. & Caldwell, I.C. A Pragmatic Approach to Distributing Ada for Transputers. In Bishop, J., Ed., *Distributed Ada: Developments and Experiences (Proceedings of the Distributed Ada '89 Symposium; Southampton, UK; 11-12 Dec. 1989)*, Cambridge University Press, Cambridge, UK, 1990, pp. 200-221.

89. Donohoe, P. *A Survey of Real-Time Performance Benchmarks for the Ada Programming Language.* Tech. Rept. SEI-87-TR-28, ADA200608, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

90. Donohoe, P. *Ada Performance Benchmarks on the Motorola 68020.* Tech. Rept. CMU/SEI-87-TR-40, ADA200610, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

91. Donohoe, P. *Ada Performance Benchmarks on the MicroVAX II: Summary and Results.* Tech. Rept. CMU/SEI-87-TR-27, ADA200607, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

92. Donohoe, P. *Hartstone Benchmark Results and Analysis.* Tech. Rept. SEI-90-TR-7, ADA226817, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

93. Donohoe, P. *Hartstone Benchmark User's Guide, Version 1.0.* Tech. Rept. SEI-90-UG-1, ADA235740, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

94. Doscher, H. An Ada Case Study in Cellular Telephony Testing Tools. In Lynch, B., Ed., *Ada: Experience and Prospects, Proceedings of the Ada-Europe International Conference (Dublin, 12-14 June 1990)*, Cambridge University Press, Cambridge, 1990, pp. 24-35.

95. Engle, C., Firth, R., Graham, M.H., Wood, W.G. *Interfacing Ada and SQL.* Tech. Rept. CMU/SEI-87-TR-48, ADA199634, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, December 1987.

96. Engle, C.B., Jr. The Myth of Portability in Ada. Proceedings of the Third Annual Ada Software Engineering Education and Training (ASEET) Symposium, June, 1988, pp. 219-227.

97. Engle, C. B., Jr. (Ed.) *Proceedings TRI-Ada'90.* ACM, New York, 1990.

98. Engle, C. B., Jr. *Automated and Semi-Automated Adaptive Protocols for Real-Time Rate Monotonic Scheduling.* Ph.D. Th., Polytechnic University, Brooklyn, NY, January 1992.

99. Engle, C. B., Jr. (Ed.) *Proceedings TRI-Ada'92.* ACM, New York, 1992.

100. Evaluation and Validation Team. *Ada Programming Support Environment (APSE) Evaluation and Validation (E&V) Team.* Ada Joint Program Office, Washington, D.C., 1991.

101. Federal Aviation Administration. *Advanced Automation Systems Experiences with Ada.* Tech. Rept. AAP-1, Federal Aviation Administration, U.S. Dept. of Transportation, Washington, D.C., May 1991.

102. Feldman, M.B. & Koffman, E.R. *Ada: Problem Solving and Program Design.* Addison-Wesley, Reading, MA, 1992.

103. Firesmith, D.G. Should the DoD Mandate a Standard Software Development Process? Proceedings of Joint Ada Conference 1987, March, 1987, pp. 159-167. Also published in Defense Science and Electronics, Vol 6 Number 4, April 1987, pp. 60-64.

104. Firesmith, D.G. "Mixing Apples and Oranges or What is an Ada Line of Code Anyway?". *Ada Letters* (September/October 1988), 110-112.

105. Firth, R., Mosley, V., Pethia, R., Roberts, L. & Wood, W. *A Guide to the Classification and Assessment of Software Engineering Tools.* Tech. Rept. CMU/SEI-87-TR-10, ADA213968, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1987.

106. Firth, R., Wood, W., Pethia, R., Roberts, L., Mosley, V. & Dolce, T. *A Classification Scheme for Software Development Methods.* Tech. Rept. CMU/SEI-87-TR-41, ADA200606, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1987.

107. Foreman, J. Building Software Tools in Ada - Design, Reuse, Productivity, Portability. Presented at AdaJUG, Baltimore, July 1985.

108. Foreman, J. T., & Engle, C. B., Jr. (Ed.) *Proceedings TRI-Ada'89.* ACM, New York, 1989.

109. Foreman, J. & Goodenough, J. *Ada Adoption Handbook: A Program Manager's Guide.* Technical Report CMU/SEI-87-TR-9, ADA182023, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1987.

110. Fowler, P., and Rifkin, S. *Software Engineering Process Group Guide.* Technical Report CMU/SEI-90-TR-24, ADA235784, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1990.

111. Fox, E. A. *Resources in Ada.* ACM Press, New York, 1990.

112. General Accounting Office. *Information Resources: Summary of Federal Agencies' Information Resources Management Problems.* Fact Sheet GAO/IMTEC-92-13FS, General Accounting Office, Washington, D.C., February 1992.

113. Godfrey, S. & Brophy, C. *Implementation of a Production Ada Project: The GRODY Study.* Tech. Rept. NASA-TM-103305 (Software Engineering Laboratory Series SEL-89-002), NASA Goddard Space Flight Center, Greenbelt, MD, September 1989. NTIS N90-21544.

114. Gothe, M.C., Wengelin, D., & Asplund, L. "The Distributed Ada Run-Time System DARTS". *Software - Practice and Experience 21*, 11 (November 1991), 1249-1263.

115. Graham, M.H. *Guidelines for the Use of SAME.* Tech. Rept. CMU/SEI-89-TR-16, ADA228027, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1989.

116. Grau, J. K. & Gilroy, K. A. "Compliant Mappings of Ada Programs to the DoD-STD-2167 Static Structure". *Ada Letters* (March/April 1987), 73-84.

117. Gross, R. R., & Umphress, D. A. Software Engineering as a Radical Novelty: The Air Force Ada Experience. In Engle, C. B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 501-507.

118. Hammons, C. "What Is a Million Lines of Code?". *Ada Rendezvous* (Spring/Summer 1986), 23-24. Published by Texas Instruments, Military Computer Systems Department, Plano, Texas.

119. Hefley, W. E., & Martin, C. E. Software Engineering Using Ada as an Enabling Technology. In Drew, B., Powell, D., & Kinney, L., Ed., *Computers and Communications ... Shaping the Future (Proceedings of the First Annual AFCEA Midwest Regional Conference)*, AFCEA Dayton-Wright Chapter, Dayton, OH, 1990, pp. 97-102.

120. Hill, F., Mellor, S., de Nevers, K., & Shlaer, S. *Documentation Guidelines for Ada Object-Oriented Development Using DoD-STD-2167A.* Technical Report TR-L801-066 (CECOM Contract DAAB07-88-D-L801, D.O. 0008), Project Technology, Inc., Berkeley, CA, September 1989.

121. Hogan, M.O., Hauser, E.P. & Menichiello, S.P. *The Definition of a Production Quality Ada Compiler.* Tech. Rept. SD-TR-87-29, The Aerospace Corporation, 20 March, 1987. Prepared for Space Division, Air Force Systems Command under contract #F04701-85-C-0086.

122. Holibaugh, R., Cohen, S., Kang, K. & Peterson, S. Reuse: Where to Begin and Why. In Foreman, John T., & Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'89*, ACM, New York, 1989, pp. 266-277.

123. Hooper, J. W. & Chester, R. O. Software Reuse: Managerial and Technical Guidelines. In *Proceedings of the 8th Annual National Conference on Ada Technology*, ANCOST, Inc., March 1990, pp. 424-435.

124. Humphrey, W. S. *CASE Planning and the Software Process.* Technical Report CMU/SEI-89-TR-26, AD A219 066, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1989.

125. Humphrey, W. S. *Managing the Software Process.* Addison-Wesley, Reading, MA, 1989.

126. Humphrey, W. S., Kitson, D. H., and Kasse, T. C. *The State of Software Engineering Practice: A Preliminary Report.* Technical Report CMU/SEI-89-TR-1, ADA206573, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1989.

127. Humphrey, W.S.; Snyder, T.R.; Willis, R.R. "Software Process Improvement at Hughes Aircraft". *IEEE Software 8*, 4 (July 1991), 12-23.

128. IBM System Integration Division. *Software-First Life Cycle Final Definition*. STARS CDRL Document 1240-001, IBM System Integration Division, Gaithersburg, MD, January 1990.

129. IEEE. *IEEE Std 1016: Guide to Software Design Descriptions.* IEEE, New York, 1992. Technical Committee on Software Engineering of the IEEE Computer Society.

130. IEEE. *IEEE Std 1074-1991: Standard for Software Life Cycle Processes.* IEEE, New York, 1991.

131. IEEE. *IEEE Std 716-1989: C/ATLAS Test Language.* IEEE, New York, 1989.

132. IEEE. *IEEE Std 990-1987: Recommended Practice for Ada as a Program Design Language.* IEEE, New York, 1987. Technical Committee on Software Engineering of the IEEE Computer Society.

133. IIT Research Institute. *Available Ada Bindings.* IIT Research Institute, Lanham, MD, 1992. Prepared for Ada Joint Program Office, under Contract DOD-MDA-903-87-D-0056, Delivery Order 007. Available online from the Ada Information Clearinghouse as bindings.hlp.DDMMMYY, where DDMMMYY is the date of the most recent update to this file.

134. IIT Research Institute. *Catalog of Resources for Education in Ada and Software Engineering (CREASE), Version 6.0.* Ada Joint Program Office, Washington, D.C., 1992.

135. IIT Research Institute. *Test Case Study: Estimating the Cost of Ada Software Development*. IIT Research Institute, Lanham, MD, April 1989.

136. International Resource Development Inc. "European Influence Broadening in U.S. Ada Market". *Ada Data 9*, 1 (January 1991), 4-6.

137. Jablonski, J. R. *Implementing Total Quality Management: An Overview.* Pfeiffer & Company, San Diego, 1991.

138. Johnson, K., Simmons, E. & Stluka, F. *Ada Quality and Style: Guidelines for Professional Programmers, Version 2.0*. Tech. Rept. SPC-91061-N, Software Productivity Consortium, Herndon, VA, 1991. This report (SPC-91061-N), defining AJPO's suggested style guide, is available for downloading from the AdaIC electronic bulletin board and the AJPO host computer. It is available in hard copy from the Defense Technical Information Center (DTIC) and National Technical Information Service (NTIS) as document number is AD-A242-525..

139. Jones, C. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, New York, 1991.

140. Jordano, A.J. Managing New Software Technology. Presented at Ada Expo, Charleston, West Virginia, November 1986.

141. Judge, J. F. "Ada Progress Satisfies DoD". *Defense Electronics 17*, 6 (June 1985), 77-87.

142. Kane, P.T., Leuci, N.D., Reifer, D.J. A Cost Model for Estimating the Costs of Developing Software in the Ada Programming Language. In Shriver, B.D., Ed., *Proceedings of the 21st Annual Hawaii International Conference on Systems Sciences*, IEEE Computer Society, Washington, D.C., 1988, pp. Vol. II, pp. 782-790.

143. Kaplan, H. T. "The Ada COCOMO Cost Estimating Model and VASTT Development Estimates vs. Actuals". *Vitro Technical Journal 9*, 1 (Winter 1991), 48-60.

144. Keller, J. "GAO Attacks DoD Software Cost Tracking". *Military & Aerospace Electronics 3*, 7 (September 14 1992), 1, 14, 28.

145. Kemerer, C. F. "How the Learning Curve Affects CASE Tool Adoption". *IEEE Software 9*, 3 (May 1992), 23-28.

146. Kitson, D. H. & Humphrey, W. S. *The Role of Assessment in Software Process Improvement*. Technical Report CMU/SEI-89-TR-3, AD A227 426, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, December 1989.

147. Kitson, D. H., & Masters, S. *An Analysis of SEI Software Process Assessment Results: 1987-1991*. Technical Report CMU/SEI-92-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, July 1992.

148. Knapper, R. J., Linn, C. J., & Salasin, J. *Guidelines for Tailoring DOD-STD-2167A for SDS Software Development.* IDA Paper P-2018, Institute for Defense Analyses, Alexandria, VA, February 1988.

149. Kuhn, D. R. "IEEE's POSIX: Making Progress". *IEEE Spectrum 28*, 12 (December 1991), 36-39.

150. Ladden, R. M. "A Survey of Issues to Be Considered in the Development of an Object-Oriented Development Methodology for Ada". *Ada Letters IX*, 2 (March/April 1989), 78-89.

151. Lawlis, P. & Elam, T. W. "Ada Takes on Assembly — and Wins". *CrossTalk (Software Technology Support Center, Hill AFB, Ogden UT 32* (March 1992).

152. Lawson, R., Springer, M. & Howard, R. The Second Ada Project: Reaping the Benefits. Proceedings of the Fifth Washington Ada Symposium (1988), June 1988, pp. 69-76.

153. Leavitt, T. & Terrell, K. *Ada Compiler Evaluation Capability. Release 2.0*. Tech. Rpt. WL-TR-91-1069, Technical Operating Report (TOR) D500-12482-1, AD-A238 259, Weapons Lab, Kirtland AFB, NM, July 1991.

154. Leavitt, T. & Terrell, K. *Ada Compiler Evaluation Capability User's Guide, Release 2.0*. Tech. Rpt. WL-TR-91-1040, AD-A236 321, Weapons Lab., Kirtland AFB, NM, May 1991.

155. Leavitt, T. & Terrell, K. *Ada Compiler Evaluation Capability: Version Description Document, Release 2.0*. Tech. Rpt. WL-TR-91-1039, AD-A236 684, Weapons Lab., Kirtland AFB, NM., May 1991.

156. Lee, J.A.N. & Nyberg, K.A. *Strategies for Introducing Formal Methods into the Ada Life Cycle*. Technical Report SPC-TR-88-002, Software Productivity Consortium, Reston, VA, January 1988.

157. Lee, K., Rissman, M., D'Ippolito, R., Plinta, C. & Van Scoy, R. *An OOD Paradigm for Flight Simulators, 2nd Edition*. Technical Report CMU/SEI-88-TR-30, ADA204849, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1988.

158. Maher, J. H. Managing Technical Innovation. In *Software Technology Transition (Tutorial, 13th International Conference on Software Engineering, Austin, TX, May 12, 1991)*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1991, Chap. II, pp. 37-60.

159. Mansir, B. E. & Schacht, N. R. *An Introduction to the Continuous Improvement Process: Principles and Practices*. Tech. Rept. LMI-IR806R1, Logistics Management Institute, Bethesda, MD, August 1989. Also. avail. NTIS AD-A211 911.

160. Marciniak, J. J. & Reifer, D. J. *Software Acquisition Management.* John Wiley & Sons, New York, 1990.

161. Marmor-Squires, A., et al. A Risk Driven Process Model for the Development of Trusted Systems. In *Proceedings Fifth Annual Computer Security Applications Conference*, IEEE Computer Society Press, Washington, D.C., 1989, pp. 184-192.

162. Marshall, I. User Introduction to the Ada Evaluation System, Release 1, Version 1, Issue 2. September 1988.

163. Martin, R., et al. *Proceedings of the Workshop on Executive Software Issues*. Technical Report CMU/SEI-89-TR-6, ADA206779, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January 1989.

164. Martin, C. E., Hefley, W. E., Bristow, D. J., & Steele, D. J. "Team-Based Incremental Acquisition of Large-Scale Unprecedented Systems". *Policy Sciences 25* (1992), 57-75.

165. Mayer, J. H. "Aiding and ABETing a Coherent Testing Environment". *Military & Aerospace Electronics 3*, 3 (May 1992), 23-26.

166. Mayer, J. H. "Is Ada Ready for the 90s?". *Military & Aerospace Electronics 3*, 6 (August 1992), 21-24.

167. McGarry, F. E. The Economics of Software Engineering: 15 Years in the Software Engineering Laboratory. Presented at Executive Session at the ACM TRI-Ada '91 Conference.

168. McGarry, F. E., & Agresti, W. W. "Measuring Ada for Software Development in the Software Engineering Laboratory". *Journal of Systems and Software 9*, 2 (February 1989), 149-159.

169. McGarvey, R. L. ABET - A Standard for Ada in a Test Environment. In Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 463-470.

170. McQuown, K.L. Object-Oriented Design in a Real-Time Multiprocessor Environment. In Foreman, John T., & Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'89*, ACM, New York, 1989, pp. 570-588.

171. Mendal, G.O. & Bryan, D.L. *Exploring Ada, Volume 2.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

172. Mogilensky, J. Process Maturity as a Guide to Phased Ada Adoption. In *Proceedings Eighth Annual Washington Ada Symposium/Summer SIGAda Meeting*, ACM, New York, 1991, pp. 16-23.

173. Mohanty, S.N. *Ada in Mission Critical System Acquisition: A Guidebook*. Technical Report MTR-84W00189, MITRE Corporation, McLean, VA, September, 1984.

174. Moran, M.L. & Engle, C.B. The Pedagogy and Pragmatics of Teaching Ada as a Software Engineering Tool. In *Proceedings of the Fourth Annual Ada Software Engineering Education and Training Symposium*, Ada Software Engineering Education Team, Ada Joint Program Office, Washington, D.C., 1989, pp. 147-159.

175. Murphy, S. Software Engineering and Ada: Experiences on Advanced Automation System (AAS). Presented at Executive Session at the ACM TRI-Ada '91 Conference.

176. Naval Information Systems Management Center. *Department of the Navy Ada Implementation Guide, Volumes I and II (AD-A250 790 and AD-A250 791).* Naval Information Systems Management Center, Washington, D.C., 1992.

177. Nielsen, K. *Object-Oriented Design with Ada.* Bantam Books, New York, 1992.

178. Nissen, J.C.D. and Wallis, P.J.L. *Portability and Style in Ada.* Cambridge University Press, Cambridge, UK, 1984.

179. Nissen, J.C.D. & Wichmann, B.A. "Ada-Europe Guidelines for Ada Compiler Specification and Selection". *Ada Letters* (March/April 1984), 50-62. Originally published in October 1982 as a National Physical Laboratory (United Kingdom) Report.

180. Nyberg, K. A. *Ada: Sources and Resources (1991 Edition).* Grebyn Corporation, Vienna, VA, 1991.

181. Page, R.D. Holistic Case Study Approach to Ada-Based Software Engineering Training. In Foreman, John T., & Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'89*, ACM, New York, 1989, pp. 332-341.

182. Park, R. E., et al. *Software Size Measurement: A Framework for Counting Source Statements*. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1992.

183. Paulk, M.C., Curtis, B., Chrissis, M.B., et al. *Capability Maturity Model for Software*. Technical Report CMU/SEI-91-TR-24, ADA240603, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1991.

184. Payton, T. F. Domain-Specific Reuse: Vision, Strategies and Achievements. In *Proceedings STARS '91*, STARS Program, Arlington, VA, 1991, pp. 2-3 - 2-17.

185. Pennell, J. P. *Ada Program Manager Issues*. IDA Memorandum Report M-409, Institute for Defense Analyses, Alexandria, VA, December 1987.

186. Perry, D. E. "First Symposium on Environments and Tools for Ada". *Ada Letters XI*, 3 (Spring 1991).

187. Performance Issues Working Group. "Ada Performance Issues". *Ada Letters X*, 3 (Winter 1990).

188. Place, P.R.H., Wood, W.G., Luckham, D.C., Mann, W., & Sankar, S. *Formal Development of Ada Programs Using Z and Anna: A Case Study*. Technical Report CMU/SEI-91-TR-1, ADA235698, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1991.

189. Rate Monotonic Analysis for Real-Time Systems Project. *The Handbook of Real-Time Systems Analysis: Based on the Principles of Rate Monotonic Analysis.* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, forthcoming.

190. Redwine, S., et al. *DoD Related Software Technology Requirements, Practices, and Prospects for the Future*. IDA Paper P-1788, Institute for Defense Analyses, 1984.

191. Reifer, D.J. *Ada's Impact: A Quantitative Assessment*. Tech. Rept. RCI-TN-294, Reifer Consultants, Inc. Torrance, Calif., September 10, 1987. This report updates RCI-TN-255, March 1987.

192. Reifer, D. J. *Ada Education and Training Recommendations*. Tech. Rept. RCI-TN-405, Reifer Consultants, Inc., Torrance, CA, 28 July, 1989. Prepared for AIL Systems.

193. Reifer, D. J. SOFTCOST-Ada: User Experiences and Lessons Learned at the Age of Three. In Engle, C. B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 472-482.

194. Royce, W. Pragmatic Quality Metrics For Evolutionary Software Development Models. In Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 551-565.

195. Royce, W. TRW's Ada Process Model for Incremental Development of Large Software Systems. In *Proceedings 12th International Conference on Software Engineering (ICSE-12)*, IEEE Computer Society Press, Washington, D.C., 1990, pp. 2-11.

196. Royce, W. W. Managing the Development of Large Software Systems: Concepts and Techniques. In *WESCON Technical Papers, Volume 14*, WESCON, Los Angeles, CA, 1970.

197. Rozum, J. A. *Software Management Concepts for Acquisition Program Managers*. Technical Report CMU/SEI-92-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1992.

198. Sage, A. P. & Palmer, J. D. *Software Systems Engineering.* John Wiley, New York, 1990.

199. Savitch, W.J. and Petersen, C.G. *Ada: an Introduction to the Art and Science of Programming.* Benjamin/Cummings, Redwood City, CA, 1992.

200. Schlender, B. R. "How to Break the Software Logjam". *FORTUNE 120* (September 25, 1989), 100-112.

201. Seidewitz, E. Object-Oriented Programming in Smalltalk and Ada. In Meyrowitz, N., Ed., *OOPSLA '87 (Object-oriented Programming Systems, Languages, and Applications) Conference Proceedings*, ACM, New York, 1987, pp. 202-213.

202. Seidewitz, E. Thinking In Ada. Tutorial presented at Seventh (1990) and Eighth (1991) Annual Washington Ada Symposium.

203. Seidewitz, E. and Stark, M. An Object-Oriented Approach to Parameterized Software in Ada. In *Proceedings Eighth Annual Washington Ada Symposium/Summer SIGAda Meeting*, ACM, New York, 1991, pp. 62-76.

204. Sha, L., Klein, M. H., & Goodenough, J. B. *Rate Monotonic Analysis for Real-Time Systems*. Technical Report CMU/SEI-91-TR-6, ADA235641, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, March, 1991.

205. Shkapsky, J. M. Analysis of Training-Related Issues in the Transition to Ada in the DON (ADA246781). Master Th., Naval Postgraduate School,September 1991.

206. Shumate, K.C. *Understanding Ada (2nd edition).* Wiley, New York, 1989.

207. Siegel, J.A.L., Stewman, S., Konda, S., Larkey, P.D., & Wagner, W.G. *National Software Capacity: Near-Term Study*. Technical Report CMU/SEI-90-TR-12, ADA226694, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1990.

208. Smith, G. N., Cohen, W. M., Hefley, W. E., & Levinthal, D. A. *Understanding the Adoption of Ada: A Field Study Report*. Technical Report CMU/SEI-89-TR-28, ADA219188, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1989.

209. Sodhi, J. *Managing Ada Projects Using Software Engineering.* Tab Books, Blue Ridge Summit, PA, 1990.

210. Software Technology for Adaptable, Reliable Systems (STARS). *STARS Reuse Concepts, Volume I - Conceptual Framework for Reuse Processes, Version 1.0.* Informal Technical Report (CDRL 04040, Contract F19628-88-D-0031) STARS-TC-04040/001/00, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, MA, February 1992.

211. Software Productivity Consortium. *ADARTS - An Ada-Based Design Approach for Real-Time Systems, Version 1.0*. Technical Report SPC-TR-88-021, Software Productivity Consortium, Reston, VA, August 1988.

212. Software Productivity Solutions, Inc. *Ada Risk Handbook.* Naval Air Development Center, Warminster, PA, 1988.

213. *SQL Ada Module Description Language.* ISO/JTC1/SC22/WG9, May 1991.

214. Statistica, Inc. SIDPERS-3. Rockville, MD, 1991.

215. Taft, S. T. "An Overview of Ada 9X". *Communications of the ACM* (in press).

216. Softech. *The Testing of Ada Programs and Distributed Systems*. Prepared by Softech Houston office for NASA Avionics Systems Division, Research and Engineering, Johnson Space Center, January, 1986.

217. Tomayko, J. E. "Lessons Learned Teaching Ada in the Context of Software Engineering". *Journal of Systems and Software 10*, 4 (1989), 281-283.

218. Trimble, J. and King, K.C. "STARS '91". *STARS Newsletter II*, 3 (May 1992), 14-15.

219. U.S. Congress, Office of Technology Assessment. *Holding the Edge: Maintaining the Defense Technology Base.* U.S. Government Printing Office, Washington, D.C., 1989.

220. Umphress, D. A. Ada: Helping Executives Understand the Issues. In *Proceedings of the Fourth Annual Ada Software Engineering Education and Training Symposium*, Ada Software Engineering Education Team, Ada Joint Program Office, Washington, D.C., 1989, pp. 135-145.

221. Ville, C., & Bratel, A. A Real-Time Ada Design Method Based on DOD-STD-2167A. In Engle, Charles B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 130-140.

222. Weber, C. V., Paulk, M. C., Wise, C. J., & Withey, J. V. *Key Practices of the Capability Maturity Model*. Technical Report CMU/SEI-91-TR-25, ADA240604, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1991.

223. Wegner, P. Dimensions of Object-Based Language Design. In Meyrowitz, N., Ed., *OOPSLA '87 (Object-Oriented Programming Systems, Languages, and Applications) Conference Proceedings*, ACM, New York, 1987, pp. 168-182.

224. Weiderman, N.H., Borger, M.W., Cappellini, A.L., Dart, S.A., Klein, M.H., & Landherr, S.F. *Ada for Embedded Systems: Issues and Questions*. Tech. Rept. CMU/SEI-87-TR-26, ADA191096, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

225. Weiderman, N.H., Habermann, A.N., Borger, M.W. & Klein, M.H. A Methodology for Evaluating Environments. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments, January 1987, pp. 5-14.

226. Weiderman, N. H. *Ada Adoption Handbook: Compiler Evaluation and Selection.* Tech. Rept. CMU/SEI-89-TR-13, ADA207717, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 1989.

227. Weiderman, N., Donohoe, P., & Shapiro, R. Benchmarking for Deadline-Driven Computing. In Engle, C. B., Jr., Ed., *Proceedings TRI-Ada'90*, ACM, New York, 1990, pp. 254-264.

228. Wets, J. F. Thomson-CSF and Ada for ATC: An Experience of Eight Years. In Brandon, C. S., & Eustice, A. S., Ed., *Proceedings TRI-Ada '91*, ACM, New York, 1991, pp. 516-529.

229. Willis, R.R. Case History and Lessons Learned in Software Process Improvement. Presented at National Security Industrial Association 6th Annual Joint Conference and Tutorial on Software Quality and Productivity, Williamsburg, Virginia, April 17- 19, 1990.

230. Winkler, J. F. H. "A Definition of Lines of Code for Ada". *Ada Letters X*, 2 (March/April 1990), 89-94.

231. Wood, W., Pethia, R., Roberts, L. & Firth, R. *A Guide to the Assessment of Software Development Methods*. Tech. Rept. CMU/SEI-88-TR-8, ADA197416, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1988.

232. Worley, J. J., Jr. Education Necessary for Air Force Software Managers To Use the Ada Programming Language and Software Engineering Effectively (AFIT/GSS/ENG/91D-12, ADA246744). Master Th., Air Force Institute of Technology,December 1991.

233. Zarella, P. F., Smith, D. B., & Morris, E. W. *Issues in Tool Acquisition*. Tech. Rept. CMU/SEI-91-TR-8, ADA 244 292, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1991.

Acronyms

4GL	fourth-generation language
ABET	Ada-Based Environment for Test
ACEC	Ada Compiler Evaluation Capability
ACM	Association for Computing Machinery
ACVC	Ada Compiler Validation Capability
Ada	The language (named for Ada Lovelace, b. 1815)
ADA	Americans with Disabilities Act
AdaIC	Ada Information Clearinghouse
AdaJUG	Ada Joint Users Group (formerly Ada-JOVIAL Users Group)
ADARTS	Ada-based Design Approach for Real-Time Systems (ADARTS)
AEO	Ada Executive Official (see Software Executive Official)
AES	Ada Evaluation System
AFMC	Air Force Materiel Command
AI	artificial intelligence
AIWG	Artificial Intelligence Working Group
AJPO	Ada Joint Program Office
AML	ASEET Material Library
AMO	ACVC Maintenance Organization
ANSI	American National Standards Institute
API	application program interface
APSE	Ada Programming Support Environment
ARG	Ada Rapporteur Group
ARTEWG	Ada Runtime Environment Working Group
ASEET	Ada Software Engineering Education and Training
ASIS	Ada Semantic Interface Specification
ASSET	Asset Source for Software Engineering Technology
ATCCS	Army Tactical Command and Control System
ATLAS	abbreviated test language for all systems
AVF	Ada Validation Facility
AVO	Ada Validation Organization
AVS	Ada Validation Suite
BBS	bulletin board system
BIT	built-in test
C^3I	command, control, communications, and intelligence
CAI	computer-assisted instruction
CAIS	Common APSE Interface Set
CAISWG	CAIS Working Group

CASE	computer-aided software engineering
CAUWG	Commercial Ada Users Working Group
CDR	critical design review
CIFO	Catalogue of Interface Features and Options
CIM	Corporate Information Management (a DoD Initiative)
CM	configuration management
CMM	Capability Maturity Model for Software
CMS	Code Management System
COCOMO	Constructive Cost Model
COTS	commercial off-the-shelf [software]
CRAD	Contract Research and Development
CREASE	Catalog of Resources for Education in Ada and Software Engineering
CRG	Character Rapporteur Group
DARPA	Defense Advanced Research Projects Agency
DBMS	database management system
DEC	Digital Equipment Corporation
DIA	Defense Intelligence Agency
DISA	Defense Information Systems Agency
DLA	Defense Logistics Agency
DoD	Department of Defense
DSN	Defense Switched Network (formerly AUTOVON)
DSP	digital signal processing
DSSA	Domain-Specific Software Architectures
DTIC	Defense Technical Information Center
EDWG	Educational Products Working Group
Email	electronic mail
EMD	Engineering and Management Development
E&V	Evaluation and Validation
ExTRA	Extensions des services Temps Reels Ada
FAA	Federal Aviation Administration
FIPS	Federal Information Processing Standard
FSF	Free Software Foundation
FTP	File Transfer Protocol
FY	fiscal year
GKS	Graphics Kernel System
GPSS	general purpose system simulation
GUI	graphical user interface
HOL	high-order language, high-order programming language
HOOD	hierarchical object-oriented design

IEC	International Electro-Technical Commission
IEEE	Institute of Electrical and Electronic Engineers
IMSL	International Mathematical and Statistical Library
IR&D	independent research and development
IRG	Information Systems Rapporteur Group
ISO	International Standards Organization
MIL-STD	military standard
MIPS	millions of instructions per second
MIS	management information systems
MMS	Module Management System
NASA	National Aeronautics and Space Administration
NATO	North Atlantic Treaty Organization
NIST	National Institute of Standards and Technology
NRG	Numerics Rapporteur Group
NSA	National Security Agency
NTIS	National Technical Information Service
NUMWG	Numerics Working Group
OBJWG	Object-Oriented Working Group
OOD	object-oriented design, object-oriented development
OOP	object-oriented programming
OORA	object-oriented requirements analysis
OS	operating system
OSD	Office of the Secretary of Defense
OSF	Open Software Foundation
OSI	open systems interconnection
P^3I	preplanned product improvement
PAMELA	process abstraction method
PC	personal computer
PDL	program design language
PDSS	post-deployment software support
PHIGS	Programmer’s Hierarchical Interactive Graphics Standard
PIWG	Performance Issues Working Group
POSIX	Portable Operating System Interface
RAASP	Reusable Ada Avionics Software Packages
RAPID	Reusable Ada Products for Information Systems Development
REUSEWG	Reuse Working Group
RISC	reduced instruction set computer
RMA	rate monotonic analysis
ROM	read-only memory

RRG	Real-time Rapporteur Group
SAME	SQL Ada Module Extensions
SAMeDL	SQL Ada Module Description Language
SCCS	Source Code Control System
SDIO	Strategic Defense Initiative Organization
SDP	software development plan
SDS	strategic defense system
SDSAWG	Software Development Standards and Ada Working Group
SEE	software engineering environment
SEI	Software Engineering Institute
SEO	Software Executive Official
SEPG	Software Engineering Process Group
SIGAda	Association for Computing Machinery Special Interest Group on Ada
SLOC	source lines of code
SQL	Structured Query Language
SRG	SQL Rapporteur Group
SSTDWG	Secondary Standards Working Group
STARS	Software Technology for Adaptable, Reliable Systems
STSC	Software Technology Support Center
TQL	total quality leadership
TQM	total quality management
UIMS	user interface management system(s)
URG	Uniformity Rapporteur Group
USAF	United States Air Force
USN	United States Navy
UUT	Unit Under Test
VSR	Validation Summary Report
XRG	Ada9X Rapporteur Group

Index

Table of Contents

List of Figures

List of Tables