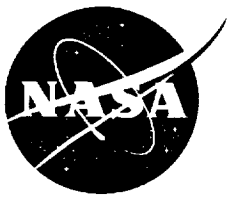
CR-189412

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-95-001

IMPACT OF ADA AND OBJECT-ORIENTED DESIGN IN THE FLIGHT DYNAMICS DIVISION AT GODDARD SPACE FLIGHT CENTER

MARCH 1995



(NASA-CR-189412) IMPACT OF Ada AND OBJECT-ORIENTED DESIGN IN THE FLIGHT DYNAMICS DIVISION AT GODDARD SPACE FLIGHT CENTER (NASA. Goddard Space Flight Center) 92 p

N95-28807

Unclas

G3/61 0053150

National Aeronautics and Space Administration

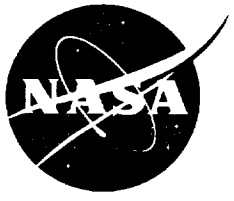
Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-95-001

IMPACT OF ADA AND OBJECT-ORIENTED DESIGN IN THE FLIGHT DYNAMICS DIVISION AT GODDARD SPACE FLIGHT CENTER

MARCH 1995



National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Foreword

The **Software Engineering Laboratory** (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on the process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The major contributors to this document are

Sharon Waligora, Computer Sciences Corporation

John Bailey, Software Metrics, Inc.

Mike Stark, NASA/Goddard Space Flight Center

The SEL is accessible on the World Wide Web at

http://groucho.gsfc.nasa.gov/Code_550/SEL_hp.html

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

Contents

Illustrations

Figures

Tables

Executive Summary

Beginning in 1985, the Flight Dynamics Division (FDD) at NASA's Goddard Space Flight Center began investigating Ada and, shortly thereafter, object-oriented design (OOD) as means of improving its products and reducing development costs for its satellite flight dynamics software systems. The FDD's intention was to become an Ada development "shop" within 10 years. This decision was based on widespread opinion in the software engineering community, particularly among U.S. Government agencies, that Ada was "more than just another programming language," that this language, in fact, represented a significant advance in software engineering technology that would lead to better products and a more disciplined practice of software engineering. Ada had been designed by the Department of Defense with the goal of providing a common language that would support the portability of programs, tools, and personnel across many projects. Another goal was to provide, in Ada, a tool beneficial for large-system development and long-term maintenance.

The Software Engineering Laboratory (SEL), which facilitates software process improvement within the FDD through an organized measurement, research, and technology infusion program, selected Ada as one of several software engineering technologies available at that time that had potential for significantly improving the local software process and products. During its initial experimentation with the language, the SEL chose to combine Ada with OOD to extend its impact throughout the full software development life cycle and to ensure that new design approaches would be explored.

The FDD's investigation of Ada/OOD was conducted as a series of experimental projects and deliverable Ada systems. Ada experiments and projects were assigned specific goals addressing different aspects of software development, such as design concepts, software reuse, cost and schedule adherence, and system performance. Progress toward these goals was guided and monitored by the SEL, and documented in study reports summarizing the results and lessons learned from each of the Ada experiences. Project characteristic data for the Ada systems were collected and stored in the SEL data base along with data for earlier FDD projects (before 1985) and concurrent FORTRAN projects (1985–1994).

This report, commissioned by the FDD and the SEL, is the product of an in-depth investigation into the Ada experience in this organization. Conducted by an outside consultant (Software Metrics, Inc.), this investigation gathered together the sum of the research and experience described above; quantitative data (system size, effort, errors, project duration, percent reuse, and performance) for all projects, both FORTRAN and Ada, active between 1985–1994; and the opinions of FDD personnel, both those directly involved in the transition and those simply present in the environment during the period. These materials have been analyzed with a focus on the evolution of local products and processes since Ada/OOD have been in use. Significant improvements in product characteristics have been documented, as well as notable changes to the software development process. This investigation also sought and identified the reasons why, 10 years after introducing this technology with an expressed intention of fully transitioning to it, and after witnessing improvements in product and process, the FDD develops only 15–20% of its software using Ada.

Although the overall assessment of this technology has shown it to be beneficial, it is unlikely that the FDD will fully transition to Ada as its language of choice. Chief among the deterrents are the lack of mainframe development environments and the high cost of viable Ada software development environments for workstations—the two platforms on which the bulk of the FDD's systems are built. Furthermore, up until now there has been no documented reason for the FDD to abandon FORTRAN as its primary implementation language. However, some of the findings in this report, regarding maintenance and software size, show good reason to move away from FORTRAN.

The key findings and technology transfer lessons learned from the FDD's Ada experience are summarized below. Recommendations based on this assessment are made regarding the future use of Ada in this organization.

Key Findings

- Use of Ada and OOD in the FDD resulted in:
  - Increased software reuse by 300%
  - Reduced system cost by 40%
  - Shortened cycle time by 25%
  - Reduced error rates by 62%
- The experimentation with Ada/OOD served as a catalyst for many of the improvements seen in the FORTRAN systems during the same period.
- FORTRAN systems applying object-oriented concepts also showed significant improvement in reuse. Like the Ada projects, higher reuse led to reduced cycle times and lower error rates on the FORTRAN projects. However, they did not experience similar cost savings; use of Ada resulted in greater cost reductions for systems with roughly comparable levels of reuse.
- Use of Ada resulted in smaller systems to perform more functionality; while generalization increased the size of the FORTRAN systems.
- Lack of viable Ada development environments on the FDD's primary development platform severely hampered the transition to Ada.
- The high cost of Ada development environments on workstations may deter future use of Ada as the FDD transitions to open systems.
- The introduction of Ada sparked much controversy within the FDD. At this time, most of the FDD workforce is lukewarm toward using Ada, with two vocal minorities for and against its continued use. However, most personnel support the use of object-oriented techniques.

Technology Transfer Lessons Learned

- Technology insertion takes a long time, especially when several technologies are combined or when the technology affects the full development life cycle and requires a significant amount of retraining.
- Parallel development experiments are an effective way of minimizing the risk of a major new technology to the organization; however, the project using the new technology must be tightly managed to maximize value and minimize negative effects.
- First impressions are very important; be careful to understand and set realistic expectations regarding the new technology for everyone affected.
- Project personnel will focus on and meet the goals set for them at the expense of those not explicitly stated. Be careful to consider all aspects of the new technology when setting goals for pilot projects, and clearly state all goals and their relative priority.
- New technology advocates are essential to initiate and sustain the technology transfer process. However, if they are not sensitive to the needs and concerns of the organization and its developers, they will impede rather than facilitate the process.
- Initial language training is best accomplished by outside vendors. Local training should focus on how to apply the language in the local environment.

Recommendations

- The FDD should continue to use Ada whenever possible. This would include for those systems that reuse existing Ada code and any other projects (or portions of projects) that are expected to be long-lived and can be developed and deployed on an Ada-capable platform.
- The FDD should build reusable software in a language that supports object-oriented constructs and consider using specialized teams of experts to configure the reusable components for each mission. This would likely improve the efficiency of the reuse process.
- The FDD should investigate lower-cost alternative languages to support object-oriented development on workstations. However, trade-off analyses should consider the cost of development environments, the efficiency and quality of software development, and the ease and cost of long-term maintenance for the languages under consideration.

Note to Readers Outside the FDD

Because the FDD uses a single language and develops small to mid-sized systems with relatively short life spans, this organization was not able to apply Ada in the context for which it was originally designed. Hence, readers of this evaluation should bear in mind that this study reports only one experience with this technology. As the findings suggest, the language offers clear benefits and involves significant investment. The specific influential factors in any one organization (e.g., software domain, hardware environment, long-term goals) must be considered in any evaluation of Ada's applicability and effectiveness.

Section 1. Introduction

1.1 Background

In the early 1980s, the software engineering community at large had great hopes and expectations for the Ada language. Ada was considered to be more than just another programming language. Because it embodied several important software engineering principles and contained features to ensure good programming practices, its proper use was expected to lead to advances in the entire software development process. Furthermore, through increased reuse, reliability, and visibility of products developed using Ada, it was expected to reduce costs and shorten project durations and to lead to better and more manageable software products. Thus, the Ada language was perceived as a significant advance in software engineering that would lead to a more disciplined software engineering practice throughout the industry.

The Department of Defense (DoD) developed the Ada language to help solve its software "crisis" (i.e., the exponentially growing amount of software to be developed and maintained by the agency). The DoD expected that, as a language that both encouraged and supported software engineering principles and practices, Ada would facilitate improved program quality and maintainability and thereby reduce full life-cycle software development costs. The DoD also sought to curtail the proliferation of the many languages in use on its software systems; its vision was for Ada to become the standard language, commonly used across many projects, thus enabling the portability of software, tools, and personnel.

In 1984, DoD mandated that Ada be used on all of its new software development projects. Shortly thereafter, other government agencies and software companies began considering using Ada to develop large systems that were expected to have long lifetimes. For example, NASA mandated that Ada be used on the Space Station Project in 1985, and the Federal Aviation Agency selected Ada as its language of choice for its Advanced Automation System in 1987.

Beginning in 1985, the Flight Dynamics Division (FDD) at Goddard Space Flight Center (GSFC) began investigating Ada and, shortly thereafter, object-oriented design (OOD) as means of improving its products and reducing development costs for its satellite flight dynamics software systems. The FDD's intention was to become an Ada development "shop" within 10 years. The Software Engineering Laboratory (SEL), which facilitates software process improvement within the FDD through an organized measurement, research, and technology infusion program, selected Ada as one of several software engineering technologies available at that time that had potential for significantly improving the local software process and products. During its initial experimentation with the language, the SEL chose to combine Ada with OOD to extend its impact throughout the full software development life cycle and to ensure that new design approaches would be explored that would maximize use of Ada features.* Thus, the FDD, supported by the SEL, began its "transition" to Ada which, after 10 years, has resulted in only limited routine use of Ada in one application domain. The organization faced and overcame many of the technical challenges typically encountered when using new technology. However, other aspects of the technology infusion process were not (or could not have been) anticipated, such as the degree of difficulty and psychological factors involved in infusing such a complex technology, and the shift in direction that the Ada "market" would take.

1.2 Environment

1.2.1 Flight Dynamics Division

The Flight Dynamics Division spends approximately $20million per year developing and maintaining ground support software for NASA's scientific satellites. This software is typically ground-based, non-embedded, and scientific (algorithmic) in nature. Applications include spacecraft attitude determination, control, and simulation; maneuver planning; orbit determination and control; and mission analysis. Systems range in size from 30thousand source lines of code (KSLOC) to 1million SLOC and are

*Strictly speaking, Ada 83 is not an object-oriented language, because it does not support dynamic binding and inheritance. However, it does support objects, and OOD methodologies can take full advantage of Ada. Ada 95 more fully supports the object-oriented paradigm.

typically developed in FORTRAN on IBM mainframe computers; however, some of the smaller systems, such as the simulators, are developed on a VAX minicomputer. The FDD has recently begun to migrate to a distributed hardware environment using an open systems architecture.

Currently, the FDD maintains approximately 4.5 million SLOC of operational software. Approximately 150–200 software engineers develop and maintain the software, which is used to support up to five launches per year and ongoing mission support for 8–15 operational satellites.

In the FDD, hardware and system support software (operating systems, compilers, and tools) are provided institutionally. Software is usually developed and operated on the same hardware. Although this eliminates the need to port software from the development machine to the target machine, it restricts the freedom of the development organization to choose its development tools and environment. Because over 75% of the computer resources are used for spacecraft operational support in this environment, FDD hardware selection and configuration decisions tend to favor system operations over development.

1.2.2 Software Engineering Laboratory and Process Improvement

In 1977, the FDD joined with its major software contractor, Computer Sciences Corporation, and the University of Maryland's Department of Computer Science to form the Software Engineering Laboratory (SEL) with the expressed purpose of improving the way the FDD develops software.[1] Since then, the SEL has established and matured a software measurement program and developed a three-step process improvement paradigm that facilitates product-based process improvement.

The SEL's process improvement paradigm is shown in Figure 1. The first and most important step is *understanding* how an organization currently does business and what it values. This is done by characterizing the products generated and the process that is used to produce them. In the second step, *assessing*, the organization sets goals for improvement, and experiments with process changes, such as a new technology, that might help achieve its goals. This is done by introducing a process change on pilot projects, assessing its impact on the product, and refining it if necessary before selecting it for use

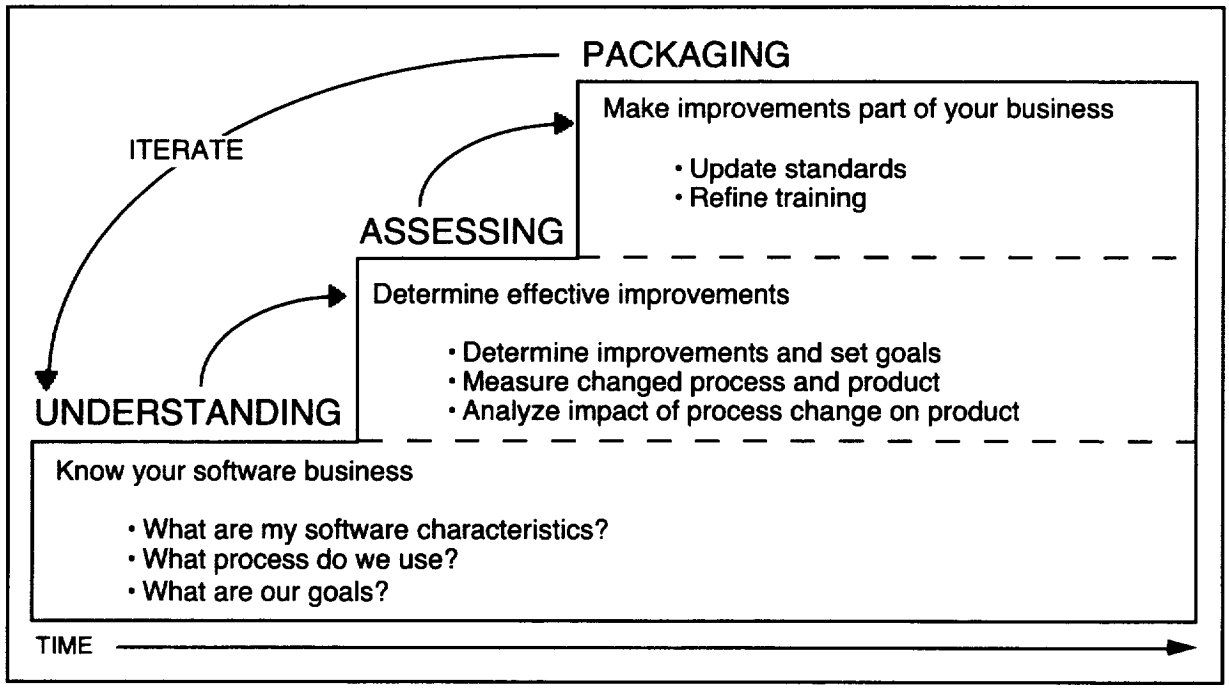


Figure 1. SEL Process Improvement Paradigm

throughout the organization. The final step is *packaging*, where the successful new technologies and procedures are integrated into the organization's standards and training program so that all projects may benefit from the changes.

Within the SEL, a group of researchers, analysts, and support personnel (separate from FDD software developers) performs process improvement activities. They collect and analyze software project measurements to produce models and standards for use by the projects. They design and monitor experiments with new technologies and modified procedures to determine their applicability to the local environment and refine/tailor them for optimum use in the FDD.

The SEL continually collects data and looks at its information to build more accurate current models. As projects and studies are completed, project data and research results are added to the baseline to increase its accuracy. In 1985, when the SEL/FDD began working with Ada, the FDD baseline was fairly well defined and a standard process had been established based on best software engineering practices, which included a well-defined life-cycle model and a disciplined approach to project management. A solid measurement program had provided data for baseline models, including

- Cost models–e.g., productivity, effort profile, code reuse, cost per change, schedule
- Reliability models–e.g., error rate, error detection rate, classes of errors (type and source)
- Process models–e.g., effort distributions by activity and by phase, software component completion profile

This baseline and the SEL's established experimentation and analysis process provided a structure for assessing the impact of new technologies such as Ada and OOD in this environment.

1.2.3 Independent Assessment

Since 1985, the FDD has completed 15 projects using Ada and OOD, while the SEL has conducted a series of experiments and studies concurrent with those projects. The SEL has produced many reports and papers about specific aspects of this work, all of which reported encouraging results. But today, nearly a decade later, the FDD still produces 80% of its software in FORTRAN, despite the initial expectation of transitioning to Ada and the positive results on Ada projects. Meanwhile other languages, such as C and C^{++}, have emerged in the intervening years as viable alternatives to both Ada and FORTRAN for scientific application programming; the FDD has recently begun using C and C^{++} on some development efforts. In addition, the organization, the standard process, and the experience of FDD personnel have continued to evolve and mature.

The SEL conducted an assessment to quantify the overall impact of Ada in the FDD, to determine why Ada has not flourished, and to recommend future directions regarding the use of Ada. In addition to comparing Ada project results with the 1985 baseline, this study compared Ada project measures with contemporaneous FORTRAN projects and assessed the applicability of Ada/OOD within the context of current organizational needs and goals. This study also attempted to capture and quantify the subjective factors affecting the technology transfer process, i.e., attitudes and beliefs held about Ada/OOD among users and others in the environment. To detect and filter any organizational bias for or against the technology, an independent consultant, Software Metrics, Inc., served as the primary investigator for this study. Preliminary results of this study were reported at the Eighteenth Annual Software Engineering Workshop in 1993.[2]

1.3 Document Organization

This report describes the SEL's approach to examining the suitability of Ada for use in the FDD and documents FDD experience using the technology on both experimental projects and on operational flight dynamics software systems. It summarizes nearly 10 years of experimentation and limited operational use of Ada, draws conclusions about why the Ada transition was less complete than expected, and offers recommendations for the role of this technology in the FDD's future. The document is organized as follows:

- Section 2 describes how Ada was introduced in the FDD and how its usage evolved. It describes
  - All of the projects on which Ada/OOD was used, both experimental and production
  - SEL studies that focused on particular organizational goals for the technology
  - Ada/OOD training offered

- Section 3 presents the results of quantitative analysis of SEL data from Ada projects. Quantitative measures of the evolving Ada experience are compared with the 1985 SEL baseline and also with the evolving FORTRAN baseline during the study period (1985–1994). Software reuse, process, cost, schedule, reliability, maintainability, and system performance are evaluated.

- Section 4 presents the qualitative data on attitudes and perceptions that were gathered by the independent assessment team. Lessons learned and factors that affected acceptance of Ada and the smoothness of the technology infusion are addressed, including vendor support.

- Section 5 summarizes Ada's overall impact on cost, schedule, reliability, and other organizational goals. It presents key findings and technology transfer lessons learned, and makes recommendations for the future use of Ada in this organization.

- Appendix A presents the data tables used in the analysis. Appendix B is a detailed analysis of the reuse approaches used on the FORTRAN and Ada projects. Appendix C includes the surveys used for collecting subjective data.

Section 2. Experience With Ada in the FDD

The FDD's activities during its transition to Ada fall into three categories that span overlapping phases thoughout the study period: experimentation and study, pilot operational use, and limited routine operational use in one application domain. Figure 2 shows the timeline for the various projects, research efforts, and studies conducted and in which activity/ transition phase they are included.

Experimentation and study have continued throughout the transition period, providing a context for investigating new approaches and resolving critical issues. The Ada work began with an experiment in 1985 that was designed to foster learning about the language and its applicability in the FDD while posing minimal risk to the operational environment. This initial experiment was conducted as a parallel effort, where two versions of the same system were developed: one in FORTRAN (operational version) and another in Ada (study version). Following this experiment, the FDD developed a series of research prototypes to investigate new ways of using Ada that would lead to future advancements (e.g., reconfigurable software). Additionally, the SEL conducted several studies to probe more deeply into issues raised by pilot projects (e.g., performance) and to better understand areas that the pilot projects would not encounter (e.g., portability).

Pilot operational use began in 1987, when the FDD began using Ada to develop small, low risk operational simulators. Each of these pilot projects focused on specific goals and contributed to the evolution of the use of Ada in the FDD, in addition to producing software systems that were used for actual mission support. Finally in 1990, the FDD began to use Ada routinely on one class of software systems, telemetry simulators.

This discussion of the FDD's Ada transition experience is organized as follows:

- Section 2.1 outlines the evolving organizational goals that provided the context for the Ada activities, and then summarizes the studies and projects, focusing on the key objectives, accomplishments, and lessons learned from each effort. To simplify the discussion, related activities are grouped.
- Section 2.2 briefly describes each project that developed operational software in Ada regardless of whether it was considered an experiment, a pilot, or routine development. Projects are in two domains: dynamics simulators and telemetry simulators. A brief discussion of GRODY, the initial experimental system, is included here because it was an important building block for design and code reuse on subsequent operational systems.
- Section 2.3 presents experience on research and development systems that have been developed to investigate new approaches.
- Section 2.4 summarizes each of the SEL studies conducted during the Ada transition, including the initial parallel development experiment (the GRODY system) and studies on reuse, portability, performance, and software size.
- Section 2.5 discusses the various approaches to training used during the transition and the effectiveness of the different methods.

2.1 Goals and Expectations

The overall goal of the FDD was to reduce development cost and cycle time for producing flight dynamics mission support systems by maximizing reuse. Ada and OOD had potential for significantly increasing reusability. In addition, the FDD was interested in adopting the high-quality software engineering practices supported and encouraged by these technologies. As local experience with Ada/ OOD grew, specific subgoals evolved within the context of the overall goal, which helped establish areas of focus for individual projects and studies. The proliferation of languages (the DoD's original concern), however, was not an issue because the FDD had always used only one or two languages for its development.

Over the past 10 years, the FDD has delivered approximately 1 million lines of Ada code. Figure 3 illustrates the growth of Ada experience in this environment. The curve shows the accumulated amount of code (in KSLOC) as each project was delivered (the time before the first project delivery is foreshortened for clarity).

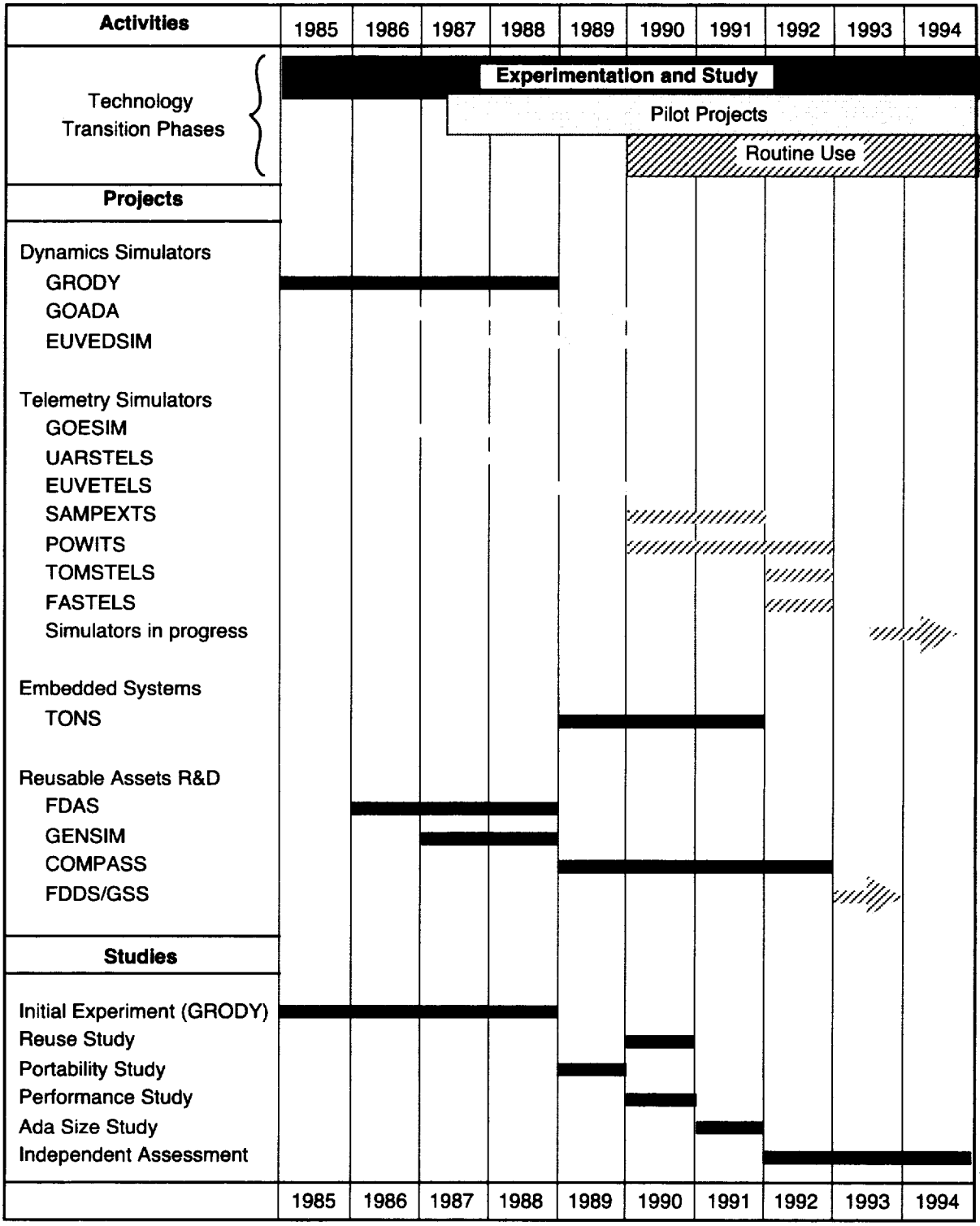


Figure 2. FDD Ada Activity Timeline

The four regions under the curve in Figure 3 give a rough approximation of the evolution of goals and objectives for the study and use of Ada in the FDD. Initially, the main concern was familiarization with the language, although the initial projects also stressed reusability as a primary objective. Soon, the focus turned to the structured generalization of systems, and the success of these generalizations led to an overall improvement in the efficiency of the Ada software development process. Recently, there has been an additional focus on optimizing the development process specifically for use with the Ada language. This optimized process has been specified and documented in a recent supplement[3] to the standard software development process guidebook used by the FDD.[4] These Ada study goals for reuse, generalization, and process provided the framework for the evolving use of Ada in this environment.

Each of the Ada projects and studies furthered the FDD's understanding of Ada and the organization's progress toward its overall goal. As short-term goals guided and focused the projects, project experience and study findings adjusted the FDD's course as it transitioned to Ada. Table 1 provides a time-ordered (by project midpoint) snapshot of the Ada project goals and key experiences.

2.2 Project Experience

By mid 1994, the FDD had completed 10 production projects in Ada using OOD. Because this technology was considered to be a radical change in this environment, the first project, GRODY, was a controlled experiment where a parallel FORTRAN system was built to the same specifications; the FORTRAN system was intended to be used operationally. This was followed by a series of projects that focused on applying the technology in new ways in continual pursuit of the organization's goals. The following sections present the FDD's project experience using Ada. Projects are grouped by application type; applications include dynamics simulators, telemetry simulators, and R&D systems.

2.2.1 Dynamics Simulators

Dynamics simulators perform closed-loop simulation of a spacecraft's attitude control system. They simulate the environment around the spacecraft, the various attitude sensors, attitude determination and attitude correction command generation based on sensor readings, and the reaction of the onboard control system to those commands. Flight dynamics analysts use these simulators to analyze the robustness of the spacecraft's attitude control system;

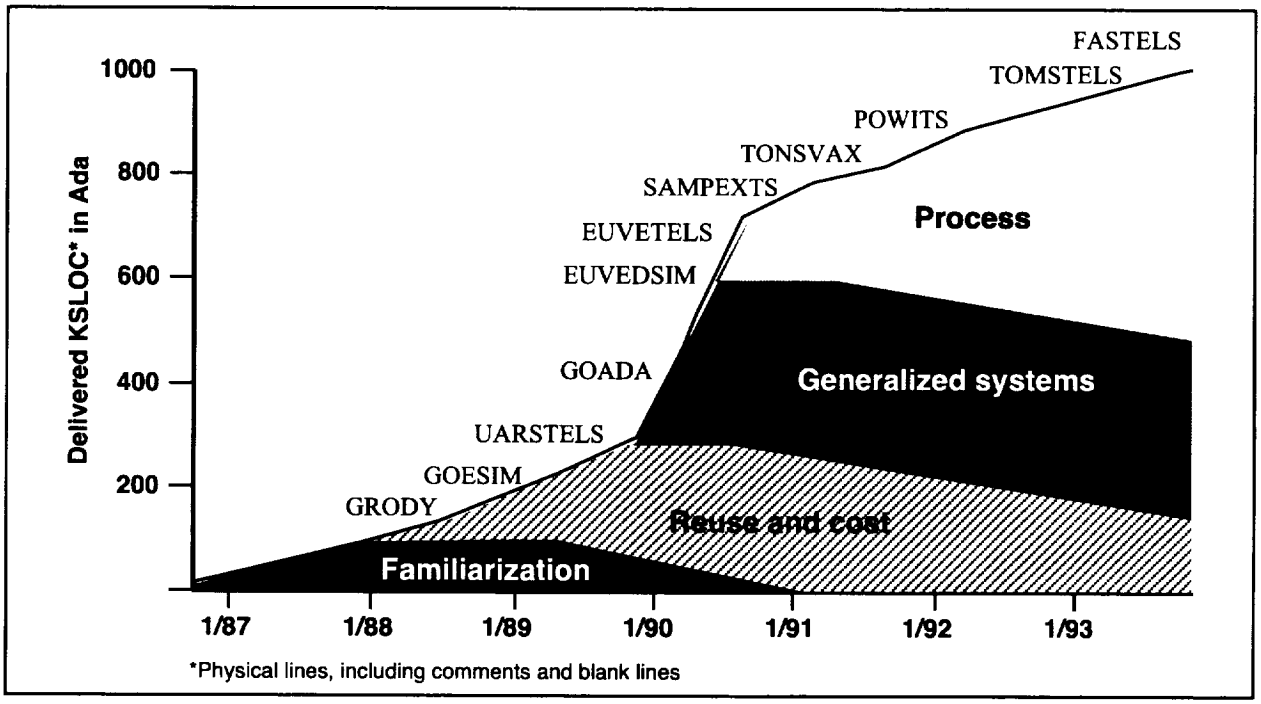


Figure 3. FDD Ada Goals and Experience

Table 1. Ada Project/Study Goals and Experience

Project or Study	Goals	Results/Key Lessons Learned
GRODY Experiment (parallel development)	Explore the Ada language; measured comparison with FORTRAN	Ada and OOD work well together; Training is required; Performance is slow
GOADA	Maximize reuse of GRODY	Reused 28% of GRODY; had serious integration problems, slow performance
GENSIM	Build generalized simulator components	Developed some generalized utilities, specifications, and concepts
GOESIM	Deliver first-of-a-kind Ada system on schedule and within budget	Delivered AdaTRAN on schedule and within budget; code not reusable
FDAS	Assess usefulness of Ada to build reconfigurable parts	Successfully used Ada generic packages to build reconfigurable components
UARSTELS	Maximize future reuse	Developed a reusable generic architecture and the corresponding reusable components
EUVETELS	Maximize verbatim reuse; reduce effort	Successfully reused 88% of UARSTELS without change
Portability Study	Investigate Ada portability by rehosting GOESIM from VAX to IBM mainframe	Mainframe compilers are immature; Ada software easily ported
EUVEDSIM	Maximize reuse; interface with FORTRAN and Ada flight code	Successfully reused 69% of GOADA; smooth integration of all components
Reuse Study	Understand what makes Ada components reusable	Identified OOD and Ada generics as reuse facilitators
TONS	Develop an embedded Ada system; performance is critical	Easily met performance requirements; encountered severe problems integrating with hardware and support software; tools available immature
Performance Study	Understand cause of performance problems	Developed Ada efficiency guidelines
SAMPEXTS	Minimize cost and schedule with high reuse	Modified process for high reuse; significantly lower cost and shorter schedule
POWITS	Maximize reuse of existing software to develop simulator for new domain; use high reuse process	Major difficulties reusing existing code for new domain; poor performance; cost and schedule overruns
COMPASS	Develop new architecture and reuse strategy for all future FDD missions	Generated specification and implementation concepts; produced generalized specifications for applications components and designed system infrastructure
Ada Size Study	Understand cost implications of Ada size variation	Preliminary cost model for Ada systems
TOMSTELS	Use high reuse process; tune performance	Routine successful project; good performance
FASTELS	Use high reuse process; tune performance	Routine successful project; good performance
FDDS/GSS	Build reconfigurable application software components based on COMPASS concepts	In progress...

they use the simulator to model the spacecraft's behavior under normal and various degraded conditions, such as failed sensors or thrusters. These simulators are used primarily to do prelaunch analysis, but also are used in emergency situations to model the spacecraft's reaction to a command sequence when a real failure occurs during the mission. Traditionally, dynamics simulators have had rather crude user interfaces. Most of the effort during development is spent on the verification of the control laws and accuracy of the hardware models.

Dynamics simulators were chosen as the first flight dynamics application to be built in Ada. They were considered a good starting point for the following reasons:

- They are relatively low-risk systems from the operational support point of view; that is, they are not used daily to provide mission operational support.
- They were usually implemented on a VAX computer where a viable Ada compiler and development tools were available.
- They are medium-sized systems by FDD standards (average size ~ 50K).
- They contain complex mathematical algorithms, which is representative of most flight dynamics systems.

To date, the FDD has developed three dynamics simulators in Ada, all based on the same system architecture with steadily increasing reuse. Key system attributes/measures are shown in Table 2.

The GRO Dynamics Simulator in Ada (GRODY) (1985-1988)

GRODY was the first Ada system developed in the FDD. This project's major purpose was to experiment with the Ada language to learn about it and to evaluate its applicability to the environment. Early in the project, the team selected OOD as an appropriate design technology to combine with Ada. Because GRODY was developed in parallel with a sister FORTRAN system (GROSS), this project did not have to produce an operational system. However, the goal of producing code potentially reusable on future deliverable systems influenced design and coding decisions.

Project-specific measures for GRODY are shown in Table 2. This project tried several alternative design approaches before settling on one. In fact, the search for an appropriate methodology led to development of a local methodology.[5] The team developed packages in parallel, but integrated them in a single build. In addition, they developed a new screen-oriented user interface that consumed a substantial portion of the project's resources. The project team made heavy use of package nesting and tasking, but used generics and typing rather sparingly.

The FDD learned several lessons from this project. Complete results of the GRODY experiment are summarized in section 2.4.1, including an extensive list of lessons learned. It is important to note a few key lessons here that greatly influenced subsequent projects:

- Heavy nesting and data coupling led to severe recompilation overhead.
- Improper or excessive use of tasking led to extremely slow performance with the available hardware resources.
- Ada source code is larger than FORTRAN equivalents when counting SLOC. The comparison of GRODY and GROSS (the operational version) source code (without adjusting for differences in functionality) revealed a 3:1 ratio of Ada to FORTRAN.

Table 2. Dynamics Simulator Project Data

Project Name	Size (KSLOC)	Overall Reuse	Verbatim Reuse	Effort (hours)	Duration (months)	Error Rate (per KSLOC)
GRODY	128K	0%	0%	23,244	39	1.8
GOADA	171K	29%	4%	28,056	34	2.4
EUVEDSIM	184K	69%	21%	20,775	23	0.7

The GOES Dynamics Simulator in Ada (GOADA) (1987-1990)

The GOES Dynamics Simulator in Ada (GOADA) project was managed as an operational software development project. Its primary goals were to produce an operational dynamics simulator using Ada and to reuse as much of GRODY as possible. The project met these goals, producing an operational system that contained 171 KSLOC, of which 29% were reused from GRODY; the entire GRODY user interface was reused. This project ended up 22% over budget and was delivered 25% late due to severe integration problems.

GOADA used the standard FDD development approach for simulators, i.e., with the exception of limiting the use of Ada tasks, developers gave little emphasis to performance issues; in fact, tradeoff decisions tended to favor fidelity over performance during design. This was a major mistake, because GOADA turned out to be one of the slowest simulators in FDD history–leaving the users and other developers with the impression that Ada was inefficient. Furthermore, although GOADA subsystems were developed in parallel, the system was integrated in a single build. This delayed the detection of interface errors and led to increased system integration cost and schedule.

The GOADA project applied lessons learned from the GRODY project. They reduced the use of tasking as their only attempt to improve performance, and they unnested all packages that were reused from GRODY to reduce recompilation overhead. This led to a lower-than-expected level of verbatim reuse (4%).

The system design made extensive use of Ada's variant record structure and strings to create a highly flexible central data structure. This allowed units to be loosely coupled, which helped facilitate recompilation of changes, but it did not allow the Ada compiler to check the data types and, as a result, many interface problems surfaced during system integration. Use of the variant records also led to excessive memory requirements on the VAX, which degraded system performance.

This project contributed several key lessons learned to the SEL's understanding of Ada and OOD:

- Performance should not be taken for granted when using new languages. Even when performance requirements are not specified, performance should be considered in the design phase and benchmarking done to understand performance impacts of design decisions.
- Unnesting significantly reduces recompilation requirements.
- Because of the size and complexity of these systems, multiple builds should be used.
- Use of variant records defeats the interface checking feature of the Ada compiler and uses excessive memory. If variant records must be used, the process should be adjusted to compensate for the lack of interface checking.

The EUVE Dynamics Simulator (EUVEDSIM) (1988-1990)

The EUVE Dynamics Simulator (EUVEDSIM) was the last dynamics simulator built in the FDD. At the end of the system testing phase of this project, the users determined that they no longer needed dynamics simulators to perform their analysis; although EUVEDSIM was thoroughly system tested it was never formally acceptance tested by the user organization. Nevertheless, this project produced an operational dynamics simulator that contained 184KSLOC.

One unique aspect of this project was that the simulator was to integrate with the actual onboard control software, part of which was implemented in Ada and part of which was implemented in FORTRAN. The EUVEDSIM project used prototyping during the design phases to learn how to interface FORTRAN and Ada components. It turned out to be fairly easy to interface between Ada and other languages on the VAX.

The other goals of this project were to continue to maximize reuse and to experiment with a build approach to eliminate the integration problems experienced on GOADA. Careful planning and strong management led the EUVEDSIM project to a successful completion. Overall reuse rose to 69%, while verbatim reuse increased to 21% because much of the code that was reused with revisions by GOADA could now be reused verbatim. Unfortunately, the variant record data structure was deeply embedded in the reused code and it came with the reuse. At this point, the FDD opted to maximize reuse rather than redesign for efficiency. The SEL, meanwhile, conducted a performance study (see section 2.4.4) to fully understand the relationship between design and coding decisions and run-time

performance, and, late in the EUVEDSIM project, some changes were made to improve performance based on the results of that study.

EUVEDSIM's major contribution to the FDD's understanding of the Ada/OOD process was its innovative build strategy. EUVEDSIM was developed and integrated in three builds, building the independent packages/subsystems first and the dependent parts last. Basically, the system was built from the inside out. This worked extremely well and integration went very smoothly. It also minimized rework and the impact of changes, because most changes occur in the algorithms, which are located in the independent packages in EUVEDSIM. Dependent parts of the code, such as the user interface, were integrated last instead of first as was common in this environment previously. In addition, code reading and unit testing standards were rigorously enforced; developers were not allowed to depend on the Ada compiler's safety net to catch their mistakes. This led to higher quality units that were easier to integrate.

This project demonstrated the following key lessons:

- A build strategy that is based on data and package dependencies works well for Ada/OOD systems. System integration went very smoothly and rework was kept to a minimum.
- Programmers discovered that the error checking provided by the Ada compiler is not a "silver bullet." Human beings still must check for indirect interfaces and accuracy of algorithms, for example. This fact was contrary to their initial belief about Ada's capabilities, which perhaps had come about from reading the literature and listening to Ada language promoters.
- Code reading is an effective way to catch problems that the compiler cannot. For example, the EUVEDSIM project carefully checked the interfaces during code reading, because they were aware of the compilation problems associated with the variant records structure.

2.2.2 Telemetry Simulators

Telemetry simulators produce simulated attitude telemetry that is used to test a spacecraft's Attitude Ground Support System (AGSS). They are batch programs that, based on a set of input parameters, model the spacecraft's attitude sensors, produce sensor readings in engineering data units, convert the data into bit streams, and pack the information into telemetry streams according to one or more telemetry formats. The size of each telemetry simulator and the amount of processing it must do is directly related to the number of telemetry formats and their associated data rate for the spacecraft. Telemetry simulators are primarily used before launch to test the AGSS and to provide simulated data for prelaunch operational simulations and training exercises. They are also used during missions to test modifications to the AGSS.

To date, the FDD has built seven telemetry simulators using Ada. All but the first one are based on the same generic architecture, which was created for the FDD's second Ada telemetry simulator project, UARSTELS. Although the first few systems were closely monitored and considered to be pilots, building telemetry simulators in Ada has become a standard way of doing business in the FDD since around 1990. Through exceptionally high levels of reuse, telemetry simulators now cost 40% less to produce, are delivered in 50% less time, and have 85% fewer errors during development when compared with the 1985 baseline for these systems. Project attributes for the telemetry simulators are shown in Table 3.

Table 3. Telemetry Simulator Project Data

Project Name	Size (KSLOC)	Overall Reuse	Verbatim Reuse	Effort (hours)	Duration (months)	Error Rate (errors/KSLOC)
GOESIM	92K	29%	12%	13,658	23	1.4
UARSTELS	68K	35%	17%	11,526	22	2.2
EUVETELS	67K	96%	88%	4,727	19	0.1
SAMPEXTS	61K	95%	85%	2,516	11	0.2
POWITS	68K	69%	39%	11,695	26	1.2
TOMSTELS	52K	97%	75%	3,839	10	0.1
FASTELS	65K	92%	64%	6,039	15	0.5

The GOES Telemetry Simulator (GOESIM) (1987-1989)

The GOES Telemetry Simulator (GOESIM) was the first telemetry simulator developed in Ada. Its goals were almost the opposite of the GRODY project goals. The GOESIM project was to demonstrate that an Ada system could be delivered on time and within budget. Tradeoffs were to favor schedule and cost over language exploration. The system was also to reuse as much code from GRODY as was possible. The project team comprised people with little or no Ada experience; a few had flight dynamics experience. However, the team received Ada training before and during project start-up.

The decision to use Ada came late on this project. Both FORTRAN and Ada were being considered up until the preliminary design review (PDR), at which time upper management committed to using Ada. This constrained the preliminary design to a FORTRAN-like structured design, which, due to budget and time constraints, was never redesigned after Ada was chosen. Thus the system made little use of Ada features and was coded in what is commonly referred to as AdaTRAN.

The project met its goals. GOESIM, comprising 92 KSLOC, was developed in two builds driven by user needs. It was delivered on schedule with only minor cost overruns well within the tolerance that was typical for flight dynamics systems. Only small utilities could be reused from GRODY, and even that was difficult, causing the GOESIM team to write extra code to interface with the reused utilities. Thus, GOESIM only achieved 29% reuse.

GOESIM contributed the following key lessons to the FDD's understanding of Ada:

- Operational schedules can be met on a first-of-a-kind Ada system, if a conservative approach is taken and experimentation is limited.
- To take full advantage of Ada, early commitment to using the language is needed; i.e., before the design phase starts.
- It is very difficult to reuse object-oriented parts in a structured design.

The UARS Telemetry Simulator (UARSTELS) (1988-1989)

The UARS Telemetry Simulator (UARSTELS) project broke new ground and laid the foundation for the FDD's future use of Ada. Project personnel, who included several experienced Ada developers, introduced a new generalized architecture that used generics extensively to facilitate future reuse. The project goals were to deliver an operational simulator on time and within budget and to continue to maximize reuse.

After analysis of GOESIM showed that there was little potentially reusable code there, the UARSTELS team proposed that they pursue the goal of maximizing reuse by focusing on *building for future reuse* rather than by *reusing* existing products. They performed domain analysis for two similar spacecraft, UARS and EUVE, for which simulators would be built almost concurrently. This helped the team identify common elements and recognize where and how a simulator should be generalized. The resulting object-oriented design isolated and packaged functionality and its data for each spacecraft element so that each element could be replaced or reused easily without affecting the rest of the system. The system was implemented using Ada generics so that only parameterized instantiation would be required to tailor packages rather than code modification. This was expected to facilitate verbatim reuse.

The project met its goals. The operational UARSTELS was delivered on time and within budget. Interestingly, UARSTELS (68 KSLOC) was smaller than GOESIM (92 KSLOC) even though UARSTELS contained more functionality. UARSTELS also required substantially more memory to execute than did GOESIM; this slowed the simulator down due to "page thrashing."

UARSTELS contributed the following lessons to the FDD's understanding of Ada/OOD:

- Use of generics leads to large memory requirements on the VAX due to the way DEC Ada implements generics.
- Heavy use of generics shrinks overall source code size, but increases executable size. SLOC

is not a good measure of functionality in Ada systems.

- Heavy use of generics makes the design harder to understand for Ada novices, including managers and users who must review it.

The EUVE Telemetry Simulator (EUVETELS) (1989-1990)

The EUVE Telemetry Simulator (EUVETELS) project's primary goal was to deliver an operational simulator on time and within budget while reusing UARSTELS extensively. Although extremely high reuse was predicted for this system, the project was planned fairly conservatively to ensure a successful delivery. The project was monitored closely to determine the impact of high reuse on effort and schedule and their allocation to life-cycle phases.

Ninety-six percent of EUVETELS (67 KSLOC) was reused from UARSTELS, of which 88% was verbatim reuse. As a result, the project achieved record high productivity and reliability rates. Due to this high level of verbatim reuse, EUVETELS created a common source code library and experimented with configuration management strategies to deal with reused code that was being changed. This was particularly challenging because the UARSTELS and EUVETELS developments were going on in parallel; thus EUVETELS was trying to reuse a moving target.

The EUVETELS demonstrated the following key lessons:

- Use of generics leads to improved verbatim reusability.
- Heavy nesting of generics makes the system complex and, therefore, hard to understand for reusers and maintainers. Better documentation is required.
- Configuration management procedures must be defined early in the project's life, when reusing a large amount of software.
- Verbatim reuse dramatically improves productivity and reliability.

The SAMPEX Telemetry Simulator (SAMPEXTS) (1990-1991)

The SAMPEX Telemetry Simulator (SAMPEXTS) project's goal was to minimize development cost while maximizing software reuse. As expected, this project achieved extremely high levels of reuse; 95% of SAMPEXTS (60 KSLOC) was reused from either UARSTELS or EUVETELS, with 85% reused verbatim. Development costs were down as well; SAMPEXTS was developed for 20% the cost of developing the system from scratch.

This project met its goal of reduced development cost by streamlining the development process. Reuse analysis was done early during the requirements definition phase and as a result the requirements and functional specifications document specified modifications to UARSTELS and EUVETELS rather than whole system functionality. Project personnel collapsed the preliminary and detailed design phases into a single design phase and held a combined PDR/critical design review (CDR). In addition, they modified the UARSTELS system description and user's guide documents during the design phase rather than generating new design documents. System testing was extremely smooth and finished ahead of schedule. Only 10 discrepancies were uncovered during the system and acceptance testing phases combined, resulting in a significantly lower system error rate of 0.2 errors per KSLOC. Disciplined code reading, inspections, and unit testing contributed to the improved reliability.

The SAMPEXTS demonstrated the following key lessons:

- A single design phase that culminates in a combined PDR/CDR works very well for high-reuse systems.
- High reuse of existing Ada software (that was designed for reuse) results in significant savings when producing very similar application software within the same problem domain.
- High reuse results in significantly fewer development errors, thus testing goes smoother and faster.

The WIND/POLAR Telemetry Simulator (POWITS) (1990-1992)

The WIND/POLAR Telemetry Simulator (POWITS) project's goal was to produce a telemetry simulator that could be used to support two missions, the Interplanetary Physics Laboratory (WIND) and the Polar Plasma Laboratory (POLAR). At the onset, the FDD expected this to be a routine high-reuse Ada project. No special goals were set. However, several

seemingly small differences became major obstacles for the team.

POWITS supported two spacecraft with spin-stabilized attitude control systems, rather than the three-axis-stabilized control system that was modeled in all of the previous Ada telemetry simulators. This essentially created a new telemetry simulator domain, which caused much of the existing software to require modification. Retrofitting the UARSTELS architecture for spin-stabilized applications was extremely difficult. In addition, the resulting design was not optimal for a spin-stabilized spacecraft, causing the system to perform poorly. The system never did meet its performance requirements.

POWITS also marked the first reuse of the UARSTELS architecture and software by a totally independent team; all of the other simulator teams included at least one member who had developed UARSTELS. As a result, the POWITS team lacked insight into the UARSTELS code, which made reusing it more difficult. This reiterated for the FDD that, although the extensive use of Ada generics contributed to high verbatim reuse, the code requires familiarity or documentation for it to be understood and reused efficiently.

The POWITS team met its goal of producing an operational simulator to support two spacecraft, but late. Delivered 7 months after its original target date, POWITS contained 68 KSLOC, of which 69% was reused and 39% reused verbatim. Most team members agree that if they had understood the existing architecture better, they would have realized the true impact of the change in attitude control systems and would have spent more time and effort investigating design alternatives, rather than trying to force fit UARSTELS. Both implementation and acceptance testing took much longer than expected: implementation due to the extent of code modifications required, and acceptance testing due to testing for two spacecraft.

POWITS contributed the following lessons to the FDD's understanding of Ada/OOD:

- Exception handling needs to be defined in detail during design. Every procedure/function should have an exception handler.
- Nested Ada generics make reusable code very hard to understand. Additional diagrams that show the overall system structure are needed to supplement the object diagrams that focus on subsystem or package composition one layer at a time.
- Ada code is not self-documenting. Novice Ada developers found the in-line commentary in the reused software to be very sparse compared with typical flight dynamics FORTRAN systems, while the code itself was harder to understand. Developers should include more comments in code that is designed for reuse.
- Code reuse cannot be assumed; it must be carefully analyzed on a case by case basis. Each project must evaluate potentially reusable systems, subsystems, and components considering the project's functional, operational, and performance requirements.

The FAST Telemetry Simulator (FASTELS) and TOMS Telemetry Simulator (TOMSTELS) (1992-1993)

By 1992, FDD's development of telemetry simulators in Ada had become routine. In addition, project plans were regularly based on expectations of high verbatim reuse. The FAST Telemetry Simulator (FASTELS) and TOMS Telemetry Simulator (TOMSTELS) projects, which began in 1992, had similar goals. Their primary goal was to produce operational telemetry simulators while maximizing reuse and guaranteeing acceptable system performance. Project schedules were now set shorter from the start based on the SAMPEXTS success with the modified process for high reuse. Using the type of attitude control system as a discriminator (i.e., spin- or three-axis-stabilized), FASTELS reused POWITS, and TOMSTELS reused UARSTELS and SAMPEXTS, respectively.

The only project concern that still remained regarding the Ada language was system performance. Higher data rates and modeling requirements were expected to severely tax the already sluggish Ada software being reused. Both projects benchmarked performance and prototyped performance enhancements during design to deal with this risk. As a result, both simulators met or exceeded their performance requirements.

Both projects delivered acceptable operational simulators on schedule. These projects both used the modified high-reuse process that collapsed preliminary and detailed design into one phase and held only one design review, confirming that the new process worked well. Developers who had not

worked on telemetry simulators previously continued to note that they had great difficulty understanding the highly general architecture and nested generic code in the reused systems.

Review of the software development history reports found no lessons learned regarding the Ada language from either project. This indicates that the use of Ada truly has become routine on telemetry simulator projects. As of late 1994, all future telemetry simulators are planned to be implemented in Ada; however, a new architecture and different design and implementation concepts will be used beginning in 1995. See the discussion of Reusable Assets Framework and Components later in this section.

2.3 Research and Development Systems

As part of the investigation of Ada and OOD, the FDD developed several R&D systems to help them understand the drawbacks and benefits of new approaches and applications. From these efforts, the organization learned about the challenges of developing embedded systems and also developed and refined an approach to creating and managingreusable assets. That research has set the direction for the FDD's formalization and exploitation of reusable components.

2.3.1 Embedded Systems

The FDD rarely builds embedded systems because its main charter is to support ground data processing. In fact, only a very small percentage of NASA's software is embedded; this is mostly onboard control software for satellites and manned spacecraft. In 1989, the FDD began developing a prototype to demonstrate the feasibility of performing spacecraft navigation computations onboard rather than on the ground in the flight dynamics facility. They chose to implement the prototype using Ada, taking advantage of the opportunity to use the language for an embedded application, the domain for which it was originally designed.

The TDRSS Onboard Navigation System (TONS) (1989-1991)

The TDRSS Onboard Navigation System (TONS) was developed in Ada on a VAX using the TARTAN Ada development environment and cross-compiler. The system was targeted to execute on dual MIL-STD-1750A microprocessors built by Texas Instruments. Both the language and the hardware were chosen for compatibility with the other onboard control systems on the EUVE spacecraft.

Performance, in terms of processing speed and memory consumption, was of great concern on this project. The design team performed a thorough compiler evaluation, examining output object code for every language construct, to determine efficiency guidelines for coding. This was extremely successful. The final product easily met its very challenging performance requirements.

However, the FDD's experience building this embedded system was discouraging. The TONS project fought a never-ending battle against subtle, undocumented, and apparently unknown incompatibilities between the 1750A hardware and the TARTAN Ada support software. Neither the 1750A nor the TARTAN Ada development tools provided a standard, mature, working environment. Because the diagnostic tools were extremely primitive, project personnel depended on hot-line vendor support to help isolate and resolve problems. However, when the team encountered significant problems, vendors were unable to provide adequate support The team had been assured at the start of the project that modifying the TARTAN "kernel" to operate in dual-processor mode was not only possible, but straightforward.

The team's development approach was first to develop the software on the VAX using the efficiency guidelines and then to test it on the VAX to verify the complex algorithms before moving it to the limited resources of the 1750A. This approach worked well. The team also developed the dual-processor communication software and modified the TARTAN kernel and tested it before porting the application software to the 1750A.

While porting the code to the 1750A, the TONS team encountered more difficulties and inconveniences involving different number precisions and restructuring the code to be able to use the TARTAN debugger. Their problems compounded when they tried to integrate all parts of the system and execute in dual-processor mode. Each processor would operate correctly when only a small driver was executing on the other processor, but the team could never get all system components to function on both processors simultaneously. The project ended up restructuring the system to operate on only one processor and delivered that system to support the

experiment. On the positive side, the team felt that working in Ada had allowed them to restructure the system very easily and quickly.

Although the failed dual-processing problem was never solved, discussions with the vendors and other experienced Ada programmers point to the large size of the TONS executable as a major contributor to the problem. Most embedded systems are small and operate within a single memory page, while TONS spanned many pages. Unfortunately, embedded navigation computations are very complex and require quite a bit of space.

2.3.2 Reusable Assets Framework and Components

The FDD developed several prototypes to gain an understanding of the importance of architecture, programming language, and library support on the reconfigurability of reusable software components. After completing the prototypes and learning from real project experience, the FDD initiated an effort to build a new project support environment and a repository for reusable application code that would facilitate the rapid construction of future flight dynamics ground systems and simulators from large-scale reusable components. To date, Ada has been the FDD's language of choice for these components, which support a broad range of flight dynamics applications.

The Flight Dynamics Analysis System (FDAS) (1986–1989)

The Flight Dynamics Analysis System (FDAS) was the FDD's first effort to use Ada to explore reconfigurable architectures. FDAS was a prototype software reconfiguration tool, which performed transaction processing from user commands to integrate and execute a library of reconfigurable parts. FDAS was very different from other flight dynamics applications, which allowed the FDD to broaden its application experience with Ada. FDAS interacted very heavily with VAX system services and had a flat, loosely coupled architecture. Because of this, the Ada packages could be developed in parallel and the system integrated and tested in a single build with ease.

Early in the project, the FDAS team grappled with the issues of how to structure the reconfigurable components so that they could be "plug compatible." After clearly defining an application structure that would meet the reconfiguration needs, the team discovered that Ada provided all of the mechanisms required to implement truly reconfigurable code. This discovery led to the UARSTELS generic architecture and formed the basis for the FDD's future work in this area.

The Generalized Simulator (GENSIM) (1987–1989)

The Generalized Simulator (GENSIM) project was a research effort to define a generalized architecture and construct generalized components that could be configured easily to produce a combined attitude dynamics and telemetry simulator. The team began with the requirements phase; based on their experience with previous simulators, they prepared functional specifications for the generalized components. Unfortunately, this project was funded at a very low level of effort and produced very little actual code. However, it furthered the FDD's understanding of reusable/reconfigurable flight dynamics application software. Specific contributions included:

- An improved set of low-level utilities
- Simplification of early systems' object states
- Generalized math specifications

The Combined Mission Planning and Attitude Support System (COMPASS) (1989–1993) and the Flight Dynamics Distributed System (FDDS) (1993–present)

The Combined Mission Planning and Attitude Support System (COMPASS) project's goal was to build a new flight dynamics project support environment and a repository for reusable application code. This project defined a distributed architecture, new user interface and executive support services, and guidelines for specifying and implementing reconfigurable flight dynamics application components. This project was to proceed in parallel with other traditional mission software development efforts. When COMPASS had developed enough support services and enough application code, it would be used to construct simulators and major ground support subsystems for mission support.

COMPASS was terminated in mid-1993 because it was too expensive to produce the software as a parallel effort without mission funding. However, the COMPASS objectives and experience were not

lost, but rather absorbed into a new conceptual framework called the Flight Dynamics Distributed System (FDDS). The major goal of these systems remains the same. The primary difference is how the projects are organized and funded. The FDDS comprises two parts, the User Interface Executive (UIX) and the Generalized Applications Support Software (GSS), which are managed independently. Although the UIX is supported with institutional funds, the GSS, which produces reconfigurable components for use on specific missions, is supported mostly by mission funding. The GSS relies heavily on the COMPASS specification and implementation concepts and is implementing all reusable applications components in Ada.

The FDDS/GSS has made significant advances in the application of Ada and OO concepts, including:

- First use of object-oriented specifications, which enables the development of classes (with the attendant cost savings) and enhances the understandability of Ada code.
- Improved use of abstract data types.
- Separation of math models (in classes) and architecture considerations (in object managers). For example, all error messages are sent via the object manager, not classes. The implication is that the classes can be reused in a different architecture without modification, but would meet the same math specification.
- Enhancement of FDD utilities for completeness, efficiency, and better abstraction.
- Run-time configuration, object allocation, and dependency setting.

2.4 Studies

Concurrent with the project experience described in the preceding paragraphs, the SEL conducted several studies to assess the risk and potential benefits of Ada/OOD and to better understand Ada-related issues as they arose through practical application of the technology. In each case, specific goals were set and the results recorded and considered when planning subsequent projects and research into the use of Ada.

2.4.1 GRODY/GROSS Parallel Development Experiment (1985–1989)

To introduce Ada into the FDD and assess its applicability, the SEL conducted a controlled study in which two dynamics simulators were developed to meet the same requirements for the Gamma Ray Observatory (GRO) mission. One system, the GRO Dynamics Simulator in FORTRAN (GROSS) was developed in FORTRAN using structured design methods, as was typical in the FDD. A second system, the GRO Dynamics Simulator in Ada (GRODY), was developed in Ada using OOD techniques. GROSS was to be used operationally and would serve as a basis for comparing both product and process measures. Both systems were built on the VAX 780 computer.

The primary goals of this experiment were to understand and characterize the Ada development process and to establish and evaluate baseline measurements for Ada development. GRODY personnel were given a substantial amount of training (see section 2.5.1 below) and were encouraged to fully exercise the language; that is, to try out all new language features that might be applicable in this environment. The FDD goal of high reuse was also emphasized and the team was encouraged to consider future reusability when designing and coding the system.

GROSS was funded, staffed, and managed as a standard FDD project. It was schedule driven and had to respond to all requirements changes. GRODY, on the other hand, was funded as a research effort. Because of this, management decisions often favored full exploration of alternative solutions to technical problems even if it resulted in schedule slippage. This led to schedule delays, with GRODY finishing 16 months after GROSS.

One of the major problems encountered by the GRODY team was the lack of available methods to transform a set of functional specifications (with an implied structured design) into an object-oriented design. The team spent a substantial amount of effort during the design phase cleansing the requirements of design implications and developing a methodology[5] for the project.

Despite the SEL's desire to keep the functionality the same in both systems, so that the relationships between FORTRAN and Ada products and project characteristics could be captured, the two systems diverged somewhat. The GRODY team designed and implemented a significantly more sophisticated user interface than typically had been supplied for dynamics simulators. Thus the code size and total effort on the Ada project ended up much higher, but perhaps would have been closer if GRODY had implemented the same user interface as GROSS. Conversely, GRODY did not have the full set of dynamic models and onboard computer models that were present in GROSS because the GRODY team was not required to respond to the many requirements changes that altered GROSS. Even with these differences, the SEL was able to get an idea of the relationship of FORTRAN to Ada parameters.

The primary results of this study are listed below:

- Training for Ada is most effective when it ensures that developers understand the software engineering principles embodied in Ada, the design methodology to be used, Ada syntax and semantics, and any vendor-specific features of the Ada environment, such as input/output details or the library management system. Managers and reviewers also need training.
- Effort distribution among life-cycle phases and activities was nearly the same for FORTRAN and Ada.
- Productivity measured as code development rate was higher in Ada, although the Ada system consumed more total effort because it was larger. GRODY's extensive new technology development and the associated learning curve drove the total effort up, thus reducing productivity.
- Reliability was lower with Ada but was considered primarily an effect of this project representing the first use of Ada in this environment.
- Ada design characteristics differed significantly from the FORTRAN/structured system. The Ada design directly reflected newer software engineering principles, such as information hiding.
- Code required more source lines with Ada, but was more readable. Counting SLOC, the Ada system was 2.5 times larger than the FORTRAN system; counting statements, it was 1.5 times larger (for similar, but not identical functionality).
- Testing showed little difference between the two languages. (This result was expected because the FDD functional testing techniques reduce the impact of the implementation language.)
- Team satisfaction was higher with Ada. At the end of the project, the Ada team requested assignment to Ada projects, and a number of the FORTRAN developers also switched to Ada.
- The General Object-Oriented Design (GOOD) methodology[5] was developed to meet the specific needs of the flight dynamics environment.

2.4.2 Reuse Study (1990–1991)

In 1990, the SEL conducted a reuse study[6] to determine reuse patterns and trends in flight dynamics systems and to determine what attributes make software components reusable. The SEL analyzed the reuse of software source code components among nine Ada projects developed in the flight dynamics environment.

SEL analysts produced six different types of reuse representations to highlight reuse among a large number of components. They discovered that the majority of Ada library units reused without change was developed specifically for flight dynamics applications rather than from the general utilities libraries that had been purchased. This contradicted the belief that purchasing a library of standard computer science components would facilitate reuse. By tracing the lineage of the highly reused components, the study provided valuable insight into the effects of unnesting, Ada generics, and OOD.

The study concluded that

- OOD significantly improved the modularity for, and level of, reuse.
- Ada generics significantly increased the level of verbatim reuse.
- Highly reusable software had been produced for telemetry simulators of three-axis-stablized spacecraft.

The study recommended that projects designing and building software "for reuse" should produce a software reuser's guide. It also recommended that

domain-specific reuse libraries be created and maintained.

2.4.3 Portability Study (1989–1990)

One of the primary goals of Ada's designers was to eliminate the proliferation of new languages and the numerous dialects of existing languages by standardizing a defined syntax. The defined syntax was expected to greatly reduce the level of effort required to port an Ada system from one environment to another. In 1989, the SEL conducted a portability study[7] to better understand the issues of portability of Ada systems. A small study team rehosted the operational GOESIM system from the VAX 8810 to an IBM 4341. A secondary goal of the study was to evaluate the suitability of the compiler available on the mainframe and tools for supporting Ada development on the mainframe.

The rehost consumed 133 staff-days over a 10-month period. Nearly 38% of the total effort was spent compiling the code and researching and fixing compilation errors. When the rehost was complete, 18% of the system had been modified and 6% had been newly created. Once the system was compiled, testing was very easy. Most of the tests passed successfully, however a few failed due to compiler anomalies. Throughout the effort, the immaturity of the compiler on the target system caused problems.

The study concluded that Ada does enhance portability. It took less effort to rehost the Ada system than a comparable FORTRAN system (based on empirical data), even with all of the compiler problems encountered. The ported system performed as expected and vendor-specific features caused fewer problems than are typical with other languages. The study team also felt that the user-defined types, in particular, made the rehost effort easier.

However, the study also concluded that the Ada compiler and development tools available on the IBM mainframe were not yet mature enough to support development of large-scale flight dynamics software systems. Debugging was difficult and expensive due to time-consuming recompilations and the lack of a debugging tool.

2.4.4 Performance Study (1990–1991)

With the introduction of Ada and OOD into the flight dynamics environment, performance surfaced as an issue. Programming in an unfamiliar language, combined with requirements for more sophisticated software systems, had highlighted the need to predict, measure, and control the run-time performance of flight dynamics systems. In 1989, the SEL initiated a study to better understand the effect of new design and implementation approaches on system performance.

The study's objectives were to determine which design and implementation alternatives lead to accelerated run-times, to identify tradeoffs necessary to achieve optimum performance, and to develop guidelines to aid future Ada development efforts in the FDD. To do this the study team performed extensive measurement and analysis of the performance of the internals of the GOADA system. They also looked at different uses of the language in small-scale benchmarks.

The study report[8] documented that incorrect design decisions were the largest contributor to poor run-time performance. It also showed that Ada compilation systems being used at that time had bugs that often contributed to poor performance. The study recommended that reused design be continually reevaluated against evolving user requirements to ensure adequate performance, and that developers use performance analysis tools to evaluate and assess compilation systems during design.

The study concluded that Ada simulators in the FDD can be designed and implemented to achieve performance comparable to existing FORTRAN simulators when performance is considered throughout the process. The study team produced a set of efficiency guidelines[9] for designing and coding Ada systems on the VAX; they are summarized in Table 4.

2.4.5 Ada Size Study (1991–1992)

By 1991, the SEL had collected measurement data from enough Ada software development projects to begin to develop an accurate cost estimation model for flight dynamics Ada systems. For years, the SEL had used software size as the basis for its cost and schedule estimation models. Each project would estimate the total number of new and reused lines of code in the system (accurately reflecting the functionality to be delivered), and then compute an adjusted size, referred to as developed lines of code (DLOC) (representing the amount of work to be done), by scaling down the reused code size by a

Table 4. Ada Efficiency Guidelines	
Requirements Analysis	• Match the data in the problem space (flight dynamics) to the appropriate data structure in the solution space.
Design	• Match the algorithm to both the data structure and the data. • Design procedures and functions for each package that map data of a general type to the data (hidden) optimal type. • Whenever the size of the structure is truly static for a particular domain, design the type as a constrained type. • Design generic components to allow users to choose between accuracy and efficiency. • Performance-critical loops should not include any string-to-enumeration conversions.
Implementation	• Looping structures should access arrays in row-major order. • Use attributes wherever possible when unconstrained structures are necessary. • Only use short-circuit control forms for performance reasons when the expression contains function calls that have no side effects.
Maintenance	• Modifications must address both the algorithmic and the data structure changes to ensure that they both still match the problem.

reuse factor. Inconsistencies in the Ada project data caused the SEL analysts to question whether size was an accurate way of representing the functionality of Ada systems. The SEL conducted the Ada Size Study[10] to answer this question and to develop a better cost estimation model for Ada project managers to use.

After characterizing the Ada development process in the FDD, the SEL concluded that the adjusted size, DLOC, was an accurate basis for estimating total project effort for Ada. However, the reuse factor, which represents the amount of work required to reuse the code, should be higher for Ada systems (0.3) than it is for FORTRAN (0.2). The study was unable to determine the cause of this difference. The study also produced a cost estimation model for Ada systems. Although the model contained different values for productivity, reuse factor, and phase distribution, the same basic SEL estimation equation worked for both FORTRAN and Ada systems. The *SEL Cost and Schedule Estimation Study Report,*[11] published in 1993, conducted a more in-depth analysis of cost and schedule trends in the FDD and offers a more thorough treatment of this topic.

2.5 Training

When infusing any new technology, training plays an important role. Training was expected to be critical for infusing Ada in the FDD because of the complexity of the Ada language and the new way of thinking required for using object-oriented techniques. Several methods were used to expose developers to Ada technology and to prepare them to use it in the FDD: commercial videotapes with outside facilitators, project-team training given by local "experts," and in-house developed college-style courses. This Ada/OOD training is discussed in the following sections, with comments on its effectiveness in this environment.

2.5.1 Initial Training

The GRODY team received extensive training in Ada before beginning the project. The training lasted approximately 6 months and was equivalent to 2 months of full-time training for each individual. The goal was to provide sufficient training in software engineering principles, language syntax, and OOD methodologies so that the team could make the best possible use of Ada, i.e., so they would produce a new, appropriate design for the Ada system as well as code it in Ada. This training is described in detail and evaluated in the *GRODY Training Evaluation*.[12]

The training was done in four parts: First, because none of the team members had previous experience with Ada, team members read Grady Booch's book *Software Engineering with Ada.* Second, the team

viewed videotaped tutorials made by Alsys, Inc., during a concentrated 40-hour period in a classroom setting where there was opportunity for discussion facilitated by a university professor. Following this, George Cherry, of Language Automation Associates, presented a 24-hour seminar on the process abstraction method design methodology. The final step, the longest and most productive part of the training, was hands-on coding of a practice problem using DEC's Ada compiler. The team spent 1336 staff hours developing an electronic message system (EMS) (5700 SLOC). Although the EMS allowed the team members to practice using Ada for standard computer science operations, it did not provide an opportunity to explore options for implementing the types of scientific functions that are common in flight dynamics software.

The GRODY team rated the discussions in the training classes and team meetings and the EMS practice problem to be the most helpful aspects of the training. The chief drawback to the practice problem was its size and simplicity. The team felt that a smaller, more complex problem requiring the use of packages and data abstraction would have been more effective for their needs. The team also listed several Ada features that were either difficult to grasp or poorly covered during the training, including input/output, tasking, generics, data types, and library units and structures. The team recommended that in-house experts prepare supplementary lectures to augment the videotapes in these areas.

2.5.2 Project-Specific Training

The early Ada projects, including GOADA, GOESIM, and FDAS, used project-specific training where the team was trained as a group using a combination of videotapes and locally prepared lectures on topics such as library management, OOD, data typing, and generics. Experienced FDD Ada developers from the GRODY project served as the trainers. This training was usually done on a part-time basis during the requirements analysis phase of the project. All currently assigned project personnel as well as those scheduled to join the project during design and implementation attended the training.

This training approach worked well during project start-up, but had its drawbacks as the projects progressed. When staffing changes became necessary, typically during the implementation and testing phases, no resource was available to train the new people coming onto the project. This was a significant disadvantage given that the entire pool of available personnel (FDD FORTRAN developers) had not yet been trained in Ada. The FDD needed a way to gradually train its existing workforce outside of the context of a specific project's immediate staffing needs.

2.5.3 Institutional Training

In 1988, the SEL developed and deployed its first Ada language course. One of the FDD Ada experts adapted the Ada language course that he taught at a local community college for the FDD environment. This course introduced the student to the Ada syntax and the software engineering principles and good practices that Ada supports. The course consisted of two 1.5 hour lectures each week for 10 weeks with a weekly hands-on homework assignment to reinforce the learning. Students were selected based on interest and high likelihood of being assigned to an upcoming Ada project. This course was generally well-accepted and interest in taking the course was very high, particularly in the contractor organization, where Ada was perceived to be an essential skill for the future.

In late 1989, the FDD brought in an outside trainer from another part of the contractor organization to provide just-in-time training for a group of new developers and managers. These people needed to be trained as replacements for a large contingent of Ada developers who had left the organization to staff new projects. The training for managers consisted of 2 days of lecture. Developer training consisted of 2 weeks of full-time hands-on training. Both courses received favorable evaluations, but were considered expensive.

When the SEL began developing a full-scale training program for FDD personnel in 1989, Ada courses were a featured product. Between 1991 and 1992, the SEL deployed this series of three Ada training courses:

- *Introduction to Ada*–Teaches the syntax and semantics of Ada and familiarizes the student with FDD Ada development tools; a series of 12 lectures over 12 weeks with weekly home-work assignments; taught by a professional teacher with limited software experience.
- *Object-Oriented Development in Ada*–Teaches Ada developers and managers the object-oriented approach to software development; covering object-oriented requirements specification, and

the analysis, design, implementation, and testing of Ada software systems; discusses FDD case studies; a series of 12 lectures over 6 weeks with weekly homework assignments; taught by a GSFC Ada expert.

- *Project Implementation with Ada*–Explains the order and relationships of the techniques, methods, tools, and products that are part of the Ada software engineering process and practices them on group projects; 6 sessions spaced out over 2 months to provide time for practice project development; taught by a contractor Ada expert.

These SEL-sponsored Ada training courses had mixed results. Course evaluation ratings ranged from highly beneficial to awful. Analysis of the data showed a high correlation between a student's course evaluation and his/her predisposition toward the language and the local FDD instructors. This was the first documented manifestation of what appeared to be a significant developer bias against Ada (see section 4.1 for more on this topic). As a result of the negative course evaluations, the SEL reevaluated its approach to institutional language training.

In the future, the FDD will use independent sources such as vendors and local colleges to teach language syntax and semantics, and in-house developed courses will be used to focus on application of the language in the local environment using the local development process and tools.

Section 3. Quantitative Analysis

The success with which the FDD met its Ada experimentation goals of increased software reuse, lower development effort, shorter cycle times, and greater software reliability was evaluated by analyzing data from contemporaneous Ada and FORTRAN flight dynamics projects.[13] Previous papers have documented improvements achieved on Ada projects over the 1985 FORTRAN baseline.[14] But, while the FDD was gradually maturing its use of Ada for satellite simulators built on DEC VAX minicomputers, the FORTRAN process used on the larger, mainframe-based projects was also evolving and improving.

This section compares the evolving Ada and FORTRAN baselines between 1985 and 1994 in each of the four experimentation goal areas (reuse, cost, schedule, and reliability) and discusses the evolving software process. It also presents a summary of the results of quantitative analyses of data on language feature use, process, and performance. Any improvements seen on the Ada projects are assessed within the context of the evolving FORTRAN baseline. Since the preliminary SEL report on this study,[2] new data have been added for completed projects in both languages, and the size and effort data have been normalized to support a more accurate comparison among projects in the two languages.

3.1 Project Data

The FDD delivered operational software to support 10 spacecraft missions from 1985 to 1994. Of these, eight missions had at least one simulator built in Ada on the VAX and an AGSS developed in FORTRAN on the IBM mainframe computer. Data from all FDD projects that produced operational software for these eight missions were examined. In particular, the series of corresponding telemetry simulators and AGSSs from the same missions were analyzed to assess the relative impact of using Ada and FORTRAN.

For each language, projects were grouped according to date (1985–1989 and 1990–1994), producing two distinct analysis periods. This division into "early" and "recent" projects occurs at a natural break in the data that corresponds with a significant increase in levels of reuse achieved and with changes in the local development process. Results were also compared with the existing SEL baseline from 1985. The complete project data used in this analysis, as well as the 1985 SEL baseline measures appear in Appendix A. Data for the MTASS/MSASS project are also included in Appendix A. This project, established in 1990, maintains and enhances FORTRAN reusable components in two controlled libraries to support missions in the two spacecraft domains: multimission three-axis-stabilized spacecraft (MTASS) and multimission spin-axis stabilized spacecraft (MSASS). (MTASS/MSASS is described in detail in the discussion of reuse approaches in section 3.2.1.)

3.1.1 Size Measures

Software size is used in these analyses as a normalizing factor when comparing productivity, reuse, error density, and process effects. The traditional measure of software size in the FDD has been source lines of code (SLOC), which counts every carriage return in the source files, including blank lines and comments. For this study, however, statement counts were chosen (i.e., the number of logical statements and declarations) because this count is not sensitive to formatting and therefore provides a more uniform indicator across the two languages both of delivered functionality and of development effort expended.[15] The average number of physical lines per statement varied over the period studied, because of the evolution of programming style and commenting conventions. Since 1985, the average number of lines per FORTRAN statement grew from 2.5 to 5 due to increased commenting, whereas the maturing Ada coding style caused the average number of lines per Ada statement to shrink from 6 to about 3.

The oldest FORTRAN code included in the study contained prologs that averaged from one-third to one-half of the total size of each subroutine, and consisted of about 20% to 30% inline comments. By the midpoint of the study period (code written around 1989), both the prolog and the inline commentary had grown so that the average number of lines per statement had increased to about 3.5. This density is representative of most of the reusable software in the FORTRAN subsystem libraries (MTASS and MSASS). Recent FORTRAN written using Cleanroom methods has even larger prologs as well

as inline commentary equal to (or greater than) the statement size, and it exhibits an overall expansion ratio of about 5 lines per statement. Recent non-Cleanroom FORTRAN exhibits a density of about 4.5 lines per statement.

On the other hand, the earliest Ada projects had extensive commentary and vertical formatting which inflated their size to an average of 6 lines per statement. A more succinct style of about 4 lines per statement had evolved by about the midpoint of the study period, when most of the reusable generic components were developed (for the UARSTELS project). Most recently, there has been a further increase in density, to about 2.5 lines per statement. The main reason for this recent increase in Ada code density was the decision to move much of the inline descriptive documentation to external references. This omission of most of the inline commentary in any new Ada code was intended to encourage reusers of the software to concentrate on understanding the semantics of the interfaces rather than studying the implementation details. The change was also intended to prevent the insertion of project-specific commentary into reusable software. Except for a standard prolog, a style similar to the documentation used for package Text_IO in the *Ada Language Reference Manual*[16] was chosen, where only brief clarification is included as inline commentary and any pertinent semantic details are contained in a companion textual reference.

3.1.2 Language Feature Usage

A 1991 report on Ada language feature usage at the FDD determined that Ada developers were attempting to use the full capability of the language.[13] The reported changes in language-feature use over time indicate that the use of Ada evolved quickly and then stabilized. Figure 4 shows four views of the evolving language usage. The figure indicates that the use of generics and strong typing increased, whereas the use of tasking decreased along with the average package size, indicators of more efficient use of Ada features. This maturation appears to have stabilized in recent years, suggesting that a standard approach has been "defined" that is appropriate for this environment and application domain.

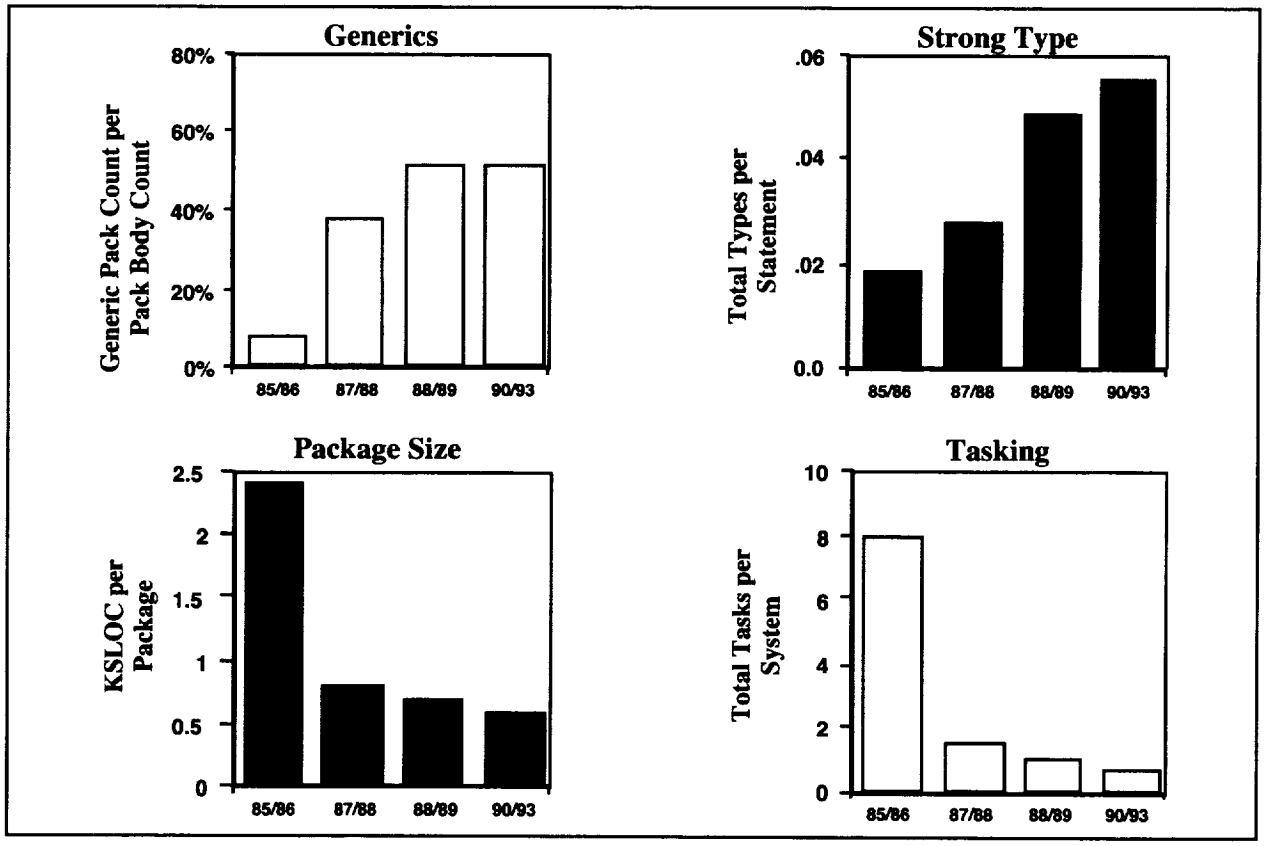


Figure 4. Maturing Use of Ada at the FDD

3.2 Reuse

During the time that Ada has been used in the FDD, there has been a considerable increase in the reuse of previously developed software on new projects. This has been achieved on all FDD projects that have applied object-oriented methods, regardless of language. Figure 5 and Figure 6 show, for Ada and FORTRAN projects, respectively, the percentage of each project that was reused without change (verbatim) from previous projects. The minimum unit of reuse is a single compilation unit; no credit is given if only a portion of a compilation unit is reused. The percentages are computed by dividing the total size of all compilation units reused verbatim by the total delivered size of the project.

Figure 5 shows a large increase in verbatim reuse in 1989 when a set of Ada generics purposely designed for reuse during the UARSTELS project was demonstrated to be sufficient to construct nearly 90% of EUVETELS, the subsequent project in the telemetry simulator domain. This level was maintained for telemetry simulators until the POWITS project (dip in the amount of reuse shown in Figure 5), when a change in the domain required that the Ada generics be modified and additional new code be developed. Specifically, the original domain where high reuse was achieved was simulation software for three-axis-stabilized spacecraft. When a spin-axis-stabilized spacecraft was simulated for the first time, a substantial drop occurred in the verbatim reusability of the library generics. This incompatibility was rectified with the creation of additional generics so that now the entire set can accommodate either a three-axis- or a spin-axis-stabilized spacecraft. The slight drop in the most recent examples of reuse to around 70%, as compared with the earlier successes that were closer to 90%, was due to performance tuning on the latest projects. Performance issues are discussed in section 3.7.

Figure 6 shows the corresponding picture of verbatim reuse on the FORTRAN projects during the same period. At its peak, the amount of verbatim reuse achieved was nearly as great as with the reusable Ada generics, and the first successes occurred at nearly the same time as the first highly successful Ada reuse. (The first high-reuse FORTRAN project, EUVEAGSS, was the corresponding ground support system for the same satellite mission as the first high-reuse Ada simulator, EUVETELS.) Again, the change in domain to spin-stabilized spacecraft caused a drop back to the low levels of reuse observed on the

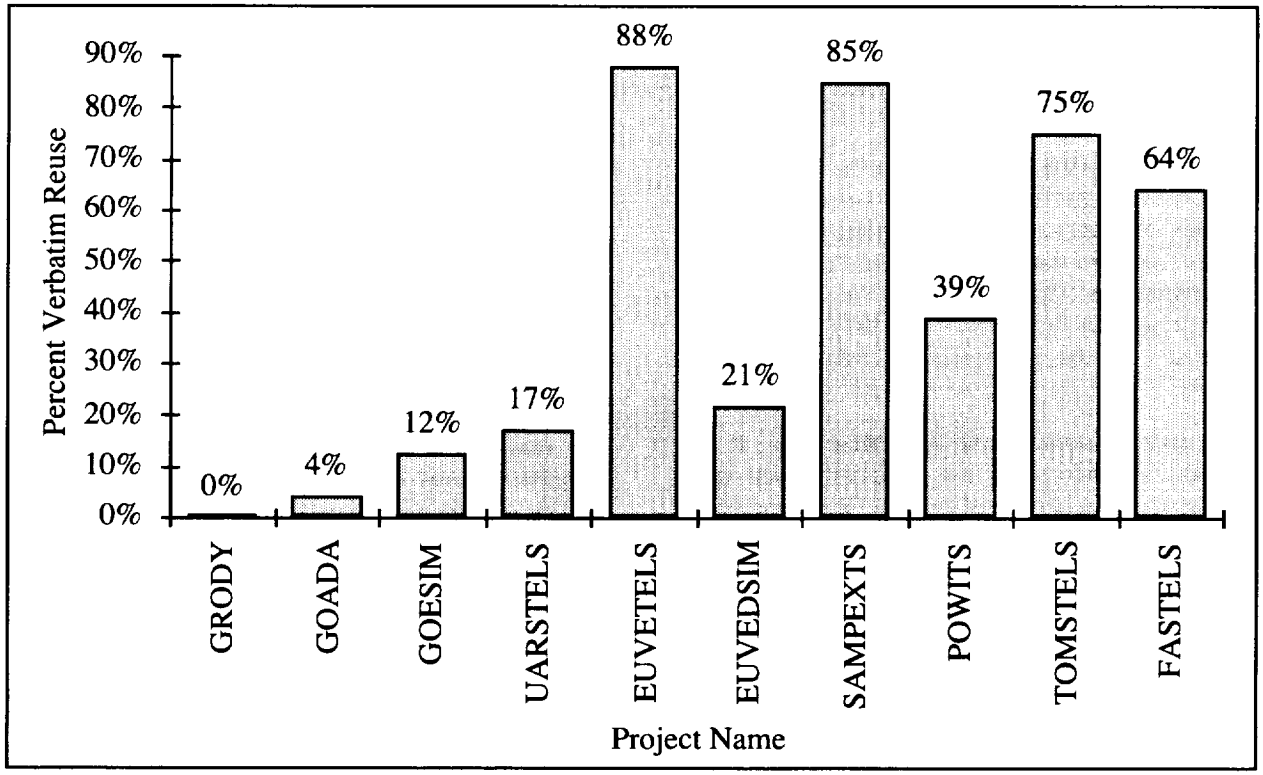


Figure 5. Verbatim Reuse Percentages for Ada Projects

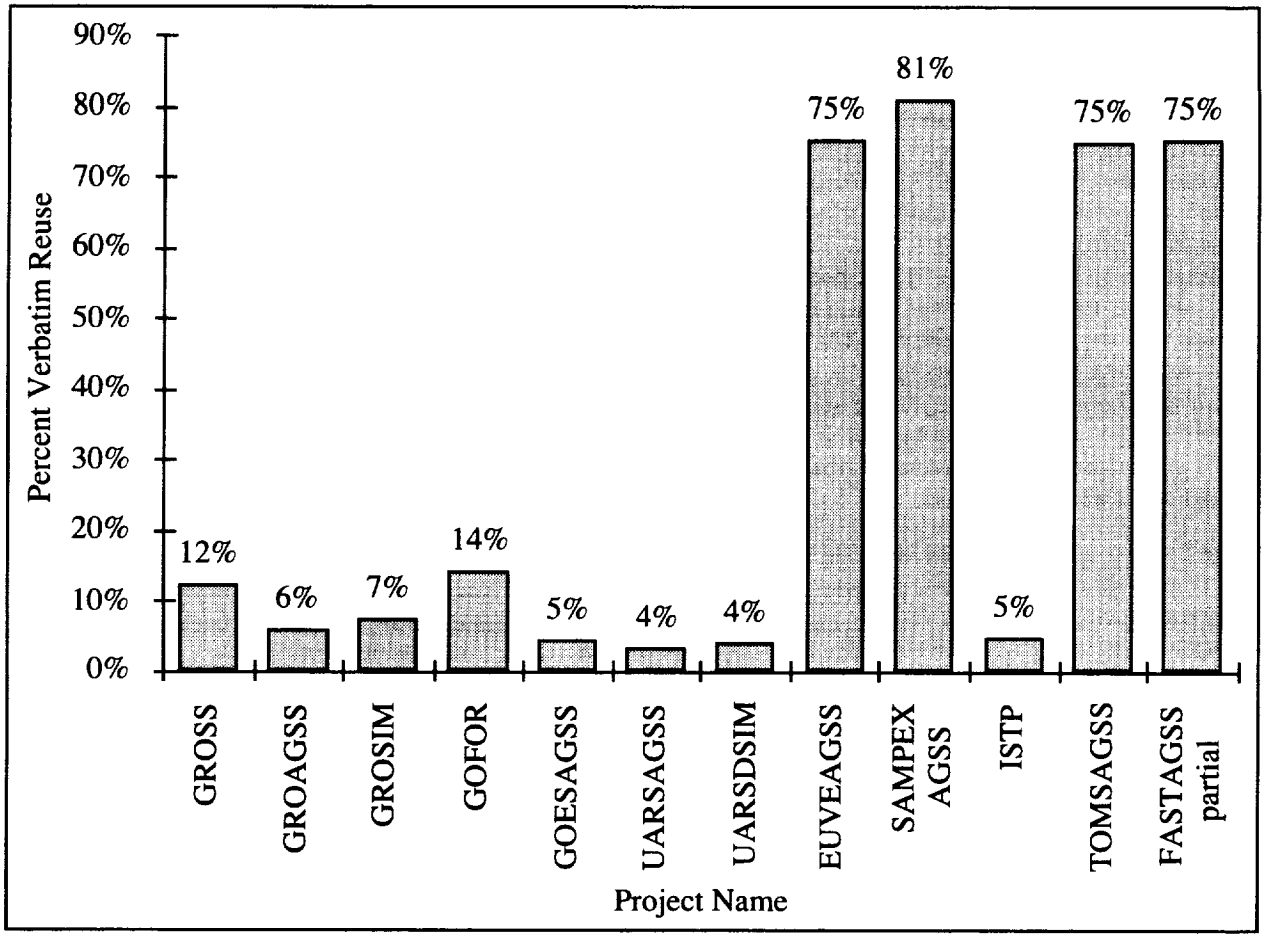


Figure 6. Verbatim Reuse Percentages for FORTRAN Projects

earlier projects in the late 1980s. In the FORTRAN case, however, the reusable components from the three-axis domain were even less suited to the spin-stabilized domain than had been the case with the Ada components. This is shown by the even greater drop in reuse on the FORTRAN ISTP system as compared with the corresponding Ada simulator, POWITS.

Before drawing conclusions from these data, it is important to understand the different reuse approaches that have been used on FORTRAN and Ada projects and to consider their effect on quantitative data. The independent assessment team thoroughly investigated the different approaches to reuse used in the two languages to determine their influence on the quantitative results. The detailed results of this research and analysis are recorded in Appendix B. The key findings of this study are discussed below.

3.2.1 Different Reuse Methods

The FDD used two different methods to manage reuse on its Ada and FORTRAN projects. This decision had more to do with the amount and expected lifetime of the software being reused, than it did with language. The AGSSs are very large and are used for many years to support active spacecraft missions; this makes strict, controlled management of the reusable code common to all AGSSs very important. Thus, the FDD chose to create a central library containing the reusable FORTRAN AGSS subsystems and to allocate a separate team to both maintain it for all active missions and modify it to support all new projects. Because of this, individual AGSS project teams are only responsible for developing new mission-specific subsystems, which they execute in concert with selected standard reusable subsystems to support a mission.

Conversely, the Ada simulators are relatively small with short operational lifetimes (on the order of months–to support prelaunch testing); this makes long-term configuration management a less important concern. Thus, Ada projects employing reusable components maintain and modify their own copy of the reusable simulator software for each mission.

Table 5 summarizes the basic differences between the reuse methods used on the FORTRAN and Ada projects. It is important to consider these differences when analyzing the quantitative data in order to differentiate, as much as is possible, between the effects of the reuse methods and the effects of using different languages.

3.2.2 Adjusting FORTRAN Measures to Compensate for Different Reuse Methods

In Figure 5 and Figure 6, both languages appear to have achieved equivalent reuse success. However, because the SEL collects reuse data only from the reusing project's perspective, this picture is somewhat misleading. The FORTRAN projects report all the subsystems that are reused from the MTASS/MSASS libraries as verbatim reuse, regardless of whether or not any units inside them need to be changed or new units need to be added by the separate maintenance team. On the other hand, the Ada projects, having responsibility for all of the software, provide a more accurate representation of verbatim reuse by reporting the reuse status of all individual units in the system. Likewise, the effort data of the Ada projects reflects the entire cost of reusing and modifying the generalized software as well as the effort required to develop new mission-specific components. But the FORTRAN project data reflects only the effort required to develop the mission-specific subsystems; the effort expended by the separate software maintenance team to understand, modify, and test the generalized reusable subsystems to meet project requirements is reported separately. Thus, the following adjustments must be made when analyzing the data.

Adjusting for FORTRAN Library Maintenance Costs

Because the high-reuse FORTRAN projects could not have delivered their systems without the services of the MTASS/MSASS library maintenance team, it is necessary to include these hours when computing the overall costs and productivities of the recent AGSS projects. This adjustment provides a fairly accurate basis for comparing the total development

Table 5. Ada vs. FORTRAN Reuse Methods

Factor	FORTRAN Systems	Ada Systems
Reusable source code management approach	Single library serves all development projects and operational missions.	Each development project and operational mission has its own copy of the reusable source.
Generalization approach for implementing reusable software	Package data and functionality together. Use case statements to handle multiple mission needs.	Use Ada generic packages to implement parameterized logic that is instantiated for specific mission at compile time via parameters.
Reuse approach	New mission-specific subsystems communicate with reused executables via data sets at run-time.	New and modified units are linked with verbatim reused units to produce project executable.
Personnel	Separate, specialized team maintains (modifies and tests) reusable code to fit new mission requirements. Project team develops new mission-specific subsystems.	Project team modifies reusable code when necessary and develops new mission-specific components.
Change philosophy	New mission requirements that affect reusable subsystems are handled by appending mission-specific 'case' logic to generalized subsystems; existing code is not touched if possible. Rigorous regression testing is done.	Mission-specific requirements are handled through parameterized generics. When modifications are necessary, the generic components are made more generalized to handle the new requirements also.

cost between languages and between the earlier and later time periods.

On the other hand, when using the data to model the cost of new, modified, or reuse-based development from the project point of view, only the reported mission-specific effort should be used. In this case, no adjustments to the FORTRAN effort data are necessary.

Adjusting verbatim reuse levels for FORTRAN projects

To clarify the relative cost of reusing externally maintained software vs. internally (project) maintained software, this analysis separates verbatim reuse into two categories:

- *Black-box reuse*–Reusable software to which the project team simply allocates requirements. When necessary, a separate team enhances and modifies the generalized subsystems to meet those requirements; the reusable software is then integrated with new mission-specific software and tested. In other words, the project team needs only to understand *what* the reusable components do, not *how* they do it.
- *White-box reuse*–Software that is reused without modification, but which the project team must read and understand, as well as test with the mission-specific software under development. In other words, the project needs to both understand what it does *and* how it does it well enough to decide if it can or should be reused.

Separating the verbatim reuse in this way allows a better approximation of the overhead involved in learning, understanding, integrating, and testing software that can be reused without change. It also provides a more equivalent basis for comparing the cost of verbatim reuse across the languages.

3.2.3 Software Size Differences Due to Generalization Approach and Language

The verbatim reuse percentages reported in the project data give the impression that the two languages are equally able to express generalized functionality. However, further investigation revealed significant differences in the structure of the reusable code and the software change philosophy used depending on the language used.

The Ada reusable architecture makes extensive use of Ada generics to provide generalized packages and procedures, which are instantiated to create mission-specific code during compilation. Because some of the generics are used repeatedly within the reusable software, the net effect was a 25% decrease (for a compression factor of .75) in the size of the code required to implement equivalent functionality when compared with earlier single-purpose mission-specific Ada code.* The FORTRAN reusable subsystems similarly used object-oriented techniques to encapsulate data and functionality into reusable components, however, the generalization was provided using parameterized case statements to determine (at run-time) which code to execute for a particular mission. Specific code was provided for each individual case. The FORTRAN type of generalization caused the reusable code size to grow. For example, an analysis of the MTASS generalized subsystems showed that they increased in size between 10% and 40% when compared with subsystems expressing similar functionality in earlier mission-specific systems; this indicates an expansion average of about 25% (or a factor of 1.25).

The maintenance approach for the reusable software, which is driven both by language and generalization approach, also affects software size. FORTRAN libraries must be continually augmented to handle new missions in their respective domains. It is the practice of the FORTRAN maintainers to augment the subsystems as necessary by adding code for any new requirements rather than by generalizing or modifying the existing code. This approach is more straightforward given the limitations of FORTRAN and it also avoids the risk of introducing errors for existing clients. However, this also causes the FORTRAN libraries to grow over time. For example, the MTASS subsystems used in both the EUVEAGSS and TOMSAGSS have grown by nearly 10% while under maintenance. Conversely, the Ada generics form a set of smaller components that requires little or no further modification to handle

*This figure is computed by first assuming that it took 15K statements of reusable UARSTELS code to deliver 110% of the GOESIM functionality, which required 18K statements. Therefore, it is expected that 110% of the GOESIM function would require 18K * 1.1 or 19.8K statements, compressing Ada-to-FORTRAN reuse to 15:19.8K or nearly .75:1. This contradicts the longstanding notion in the FDD that software size increases proportionally to functionality.

missions in either domain. The Ada developers directly handle the generics needed for each project and further generalize them only when necessary (such as by deleting unnecessary dependencies between components). Thus the size of the reusable Ada software remains roughly constant.

These size differences have the following implications:

- The generalized FORTRAN systems are on the average 25% larger than previous systems that provide similar functionality. This was considered when productivity measures were examined for this analysis. Combining the FORTRAN expansion ratio of 1.25:1 and the Ada compression factor of .75:1 results in a net difference of 1.5:1 between FORTRAN and Ada generalized software size.
- The large, and continually growing, size of the FORTRAN reusable library increases the cost of maintaining it. While 3.5K hours were required to enhance MTASS for SAMPEX (in 1991–1992), it cost between 5K and 6.5K hours per mission to enhance MTASS/MSASS for use by four mission systems in late 1992 through 1994.

3.2.4 Impact of Different Reusable Software Management Approaches

The maintenance and configuration control risks associated with maintaining separate copies of the reusable components in each client project's library never manifested on the Ada projects. It is impossible to determine from the data available whether the Ada language was influential in minimizing these risks, or whether it was due to the small size and short lifetimes of the Ada systems.

The use of single, centralized copies of the library subsystems for the FORTRAN projects and not for Ada projects introduced a mismatch that complicates direct comparison of the effort measurements for the two languages. However, the resulting data do provide some insight into the effort required to learn and modify the reusable software. Because the individual application programmers for each FORTRAN project do not have to concern themselves with the internals of the reusable subsystems, none of their effort is spent learning the reusable software. The Ada projects, on the other hand, have the burden of directly handling the reusable software, which means that Ada developers, with neither a library support team nor comprehensive documentation (as yet), must study the internals of the reusable components to understand their proper use and to determine if any enhancements are needed. The additional cost of the learning curve required to reuse software on the Ada projects can be seen when reuse is broken out into the "white-box" and "black-box" categories defined in the discussion of different reuse methods (in section3.2.2).

3.2.5 Computing the Productivity of Reuse

Conventionally in this environment, reuse is classified as either verbatim reuse or reuse with modification. Using the technique developed by Bailey,[17] individual productivities of the different categories or modes of code development/reuse in the FDD were estimated by deriving a set of simultaneous equations and then solving for the unknown productivities (see Appendix B). The effort for each project was expressed as the sum of the efforts to develop the various amounts of code in each category (new, modified, verbatim).

The best overall solutions for the productivities for new, reused with modification, and verbatim reuse for both Ada and FORTRAN code are shown in Table 6.

Table 6. FORTRAN and Ada Development Productivities*

Category of Code Reuse	FORTRAN	Ada
New Code	1.2	1.1
Reuse with Modification	2.4	1.2
Verbatim Reuse	5.5	5.0

*Statements per hour

In Table 6, the productivities for both languages are nearly identical except for the "reuse with modification" category, where the FORTRAN productivity is double that of Ada. This could indicate that FORTRAN units are easier to modify than Ada units. However, this analysis concludes that the difference actually reflects the learning curve required for reusing generic Ada code. When a project team needs to modify a part of the reusable software, additional effort is required first to understand the code and its applicability, and then to generalize it further to ensure future reusability.

Separation of the verbatim reuse category into black-box and white-box reuse for the later FORTRAN AGSS projects where MTASS and MSASS were used yielded a more stable and well-behaved set of productivity estimates for the development modes. As one might expect, the productivity for the new black-box verbatim reuse category was very high. Depending on the group of projects included in the solution, some of the analyses showed it to be essentially "infinite" (meaning that black-box statements can be "developed" for free, so the size of the reused components has little or no effect on the reusing project's cost). This means that productivityvalues for the other categories would be unaffected even if the black-box verbatim statements were eliminated from the project totals.

The productivities for the development/reuse categories with the addition of black-box verbatim reuse are shown in Table 7. There is no software in the black-box verbatim reuse category on the Ada projects.

Table 7. FORTRAN and Ada Development Productivities* Including Black-Box Reuse

Category of Code Reuse	FORTRAN	Ada
New Code	1.2	1.1
Reuse with Modification	2.4	1.2
White-Box Verbatim Reuse	4.0	5.0
Black-Box Verbatim Reuse	21.0	N/A

*Statements per hour

Reuse-library-supplied statements were included because the current reporting style is to include them in project totals. However, in the future it might make more sense to exclude them from project development estimates and reported sizes, analogous to the way the size of a math library is ignored. It would still be important to budget for the library maintenance task, however, and to understand that library maintenance remains an additional cost of delivering FORTRAN AGSS projects. Eliminating the reporting of the FORTRAN library software which masquerades as zero-cost verbatim reuse would also bring the Ada and FORTRAN reuse factors more in line with one another (see discussion of cost reduction in section 3.4).

3.3 Process Evolution

An evolving development process had as much to do with the improvements in productivity as the increases in software reuse. Without the support of an appropriate process, reuse techniques alone would not have led to the improvements observed. The software process is characterized by examining the distribution of effort across the various software development activities performed. Life-cycle activity categories include design, code, test, and "other" (e.g., management, meetings, system documentation). Figure 7 shows the average activity distributions for all Ada and FORTRAN projects during the study period. The figure shows the average percentage of staff-hours per project consumed by each activity for software projects at the FDD.

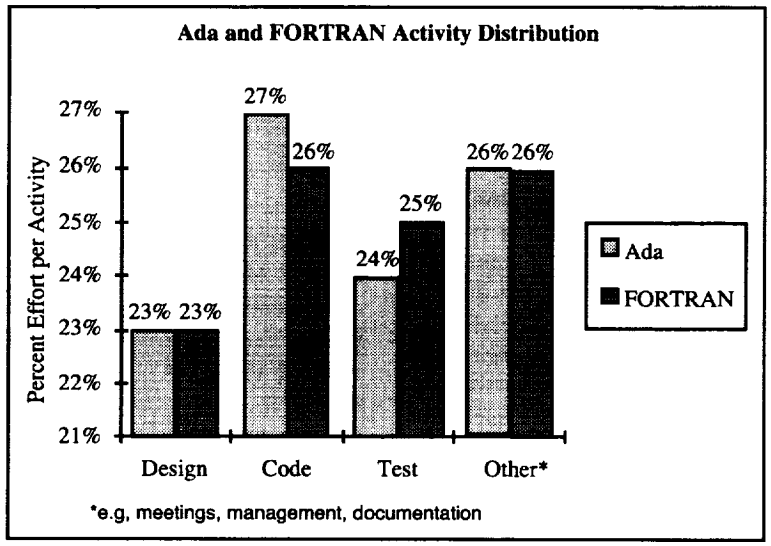


Figure 7. Activity Distribution: All Ada vs. all FORTRAN Projects

Project history reports document the fact that the software process was changing throughout the Ada study period, as seen in Figure 5 and Figure 6. These figures clearly show the points when dramatic improvements in reuse were achieved in both the Ada and the FORTRAN projects. The first Ada simulator and the first FORTRAN ground system to exhibit high reuse were both written to support the EUVE mission. Because of the nature of satellite mission support, the simulator is typically completed first so that it can be used to test the ground system. (In the case of EUVE, the Ada simulator was completed about 4 months ahead of the corresponding FORTRAN ground system.) Because these first successes with reuse almost coincided, and because they are associated with measurable changes in the development approach, the inclusion of Ada and FORTRAN projects in the "early" or "recent" set depends on whether they were completed before or after the EUVE experience. This formed a conventional reuse set and a high-reuse set of projects in each language.

The differing shapes of the early and recent activity distributions shown in Figure 8 illustrate that the more recent, high-reuse Ada projects have been, in fact, conducted using a different process than the early projects. The light bars for each activity show the averages for the first five Ada simulator projects, and the dark bars show the average effort per activity for the five recent Ada simulators that achieved higher levels of reuse. In both cases, the percentages are based on the average total effort for projects in the early set to more dramatically demonstrate the savings realized on recent projects relative to those earlier projects.

Because savings were apparent in each of the activities, it was concluded that the savings exhibited for the recent project set is not due to code reuse alone but also to the process change that came about as a byproduct of that reuse. Some of the process changes include requirements specifications expressed in terms of specific earlier system functionality, compression of PDR and CDR into one review, and reuse of baseline documentation.

Figure 9 shows the average effort by activity for the FORTRAN projects that were completed during the same period. Again, the projects are divided into an earlier group of lower reuse projects and a more recent group of higher reuse projects, and the percentages are all based on the average early project effort. As with Ada, a reduction in effort is shown for each activity when comparing low reuse with high reuse, although the net reduction is less in FORTRAN. Unlike the Ada results, however, most of the FORTRAN reduction occurs in the coding activity instead of being spread more evenly across the life cycle.

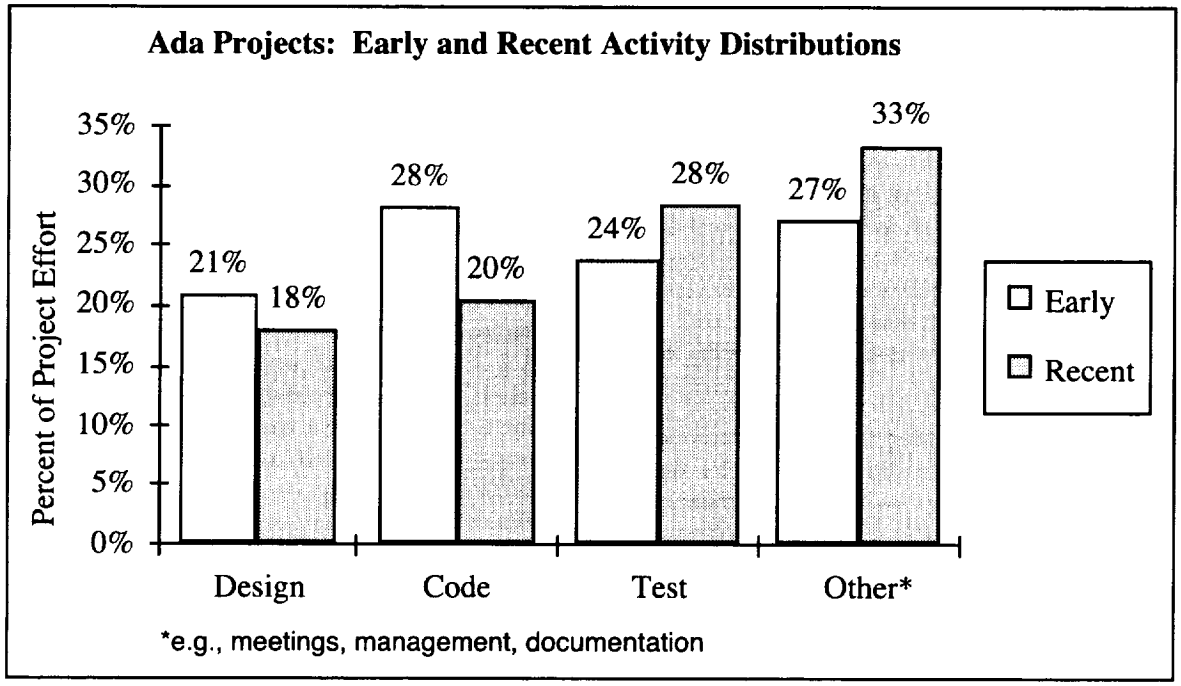


Figure 8. Activity Distribution for Ada Projects

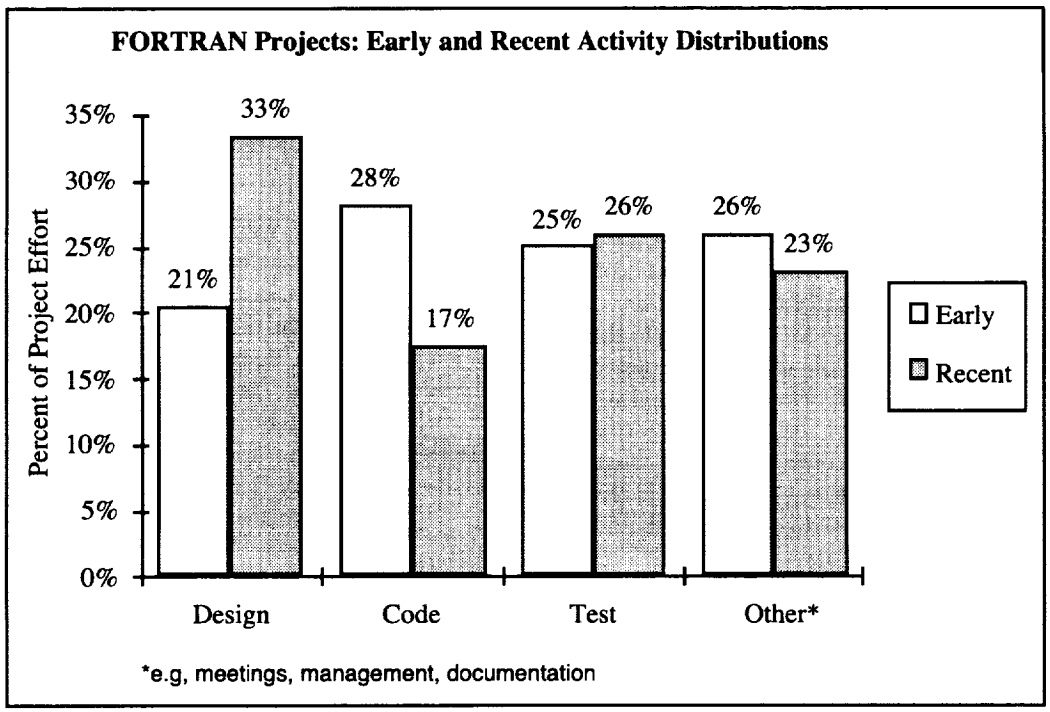


Figure 9. Activity Distribution for FORTRAN Projects

The shape of the activity distribution of the early projects in the FORTRAN set (light bars in Figure 9) is virtually identical to the activity distribution of the early projects in the Ada set (light bars in Figure 8). On the other hand, the distributions for the recent high-reuse projects differ between the languages. This suggests that the Ada and FORTRAN processes have each evolved in a different way even though they share a common ancestry. The main lessons from this illustration are that the software process matured and improved during the Ada exploration period and that this evolution affected both the Ada work and the FORTRAN work, although in different ways. Discussions later in this section will show that these process changes are also associated with a reduction in overall project cost and shortening of schedules.

The process changes associated with higher reuse in both languages were originally suggested by the developers themselves. After their initial experiences with Ada reuse and OOD, two of the chief software engineers documented their approach to capture this experience and to enable it to be more widely used by the organization.[5] This document, released in 1986 after only about 1 year of Ada experience, formed the underpinnings for the ultimately successful reuse techniques that took almost another 4 years to come into practice with the UARS and EUVE missions.

In 1990, when the use of generalized software was first shown to be possible in both Ada and FORTRAN, it became clear to the developers that the cost advantages of large-scale reuse could not be realized unless the software development process were pared down correspondingly. There were further latencies in institutionalizing these changes, however. As was noted, the EUVE projects in both Ada and FORTRAN reused large amounts of the prior UARS projects. However, a risk-reducing management decision was made to allow sufficient time and budget for the EUVE projects to be completed in a conventional fashion. It was not until the following pair of projects, for the SAMPEX mission, that the schedule and the process were actually redefined and streamlined for high reuse. Formal documentation of this new process took another 2 years.[3] The overall latency from the first attempts to incorporate reuse technology into the development process, to the adoption and formal documentation of a reuse-based process was about 8 years: 5 years to develop a reuse technology (1985–1990); 1 year to demonstrate its effectiveness; and 2 years to practice it, refine it, and document it.

3.4 Cost Reduction

The average cost to deliver a statement in each language was calculated, again adopting the distinction between conventional-reuse projects and high-reuse projects–respectively, those before and after the EUVETELS project. The left-hand side of Figure 10 shows the average cost in hours to deliver a statement of Ada, both for the early project set and the recent project set. The figure shows that the net productivity of delivering Ada software has doubled since high reuse has been achieved.

The right-hand side of the figure shows the average cost in hours to deliver a statement of FORTRAN before and after the high-reuse process. Again, there is an improvement, though not as great a reduction as in the Ada projects, particularly when the effort of the library maintenance task is computed in the total.

As recommended in section 3.2.3, the effort spent by the MTASS/MSASS library maintenance team is included in the overall cost to deliver high-reuse FORTRAN systems. The lighter of the two "recent FORTRAN" bars indicates the average cost per statement on the projects without the MTASS/MSASS contribution. The total cost, including costs for the maintenance of the independent FORTRAN reuse libraries (shown in the darker of the two bars), were computed using the total number of hours spent on all projects in each set plus, in the case of the recent FORTRAN projects, the library maintenance hours spent doing enhancements during the time period when each respective project was under development. The adjusted effort was divided by the total number of statements delivered on all application projects in each set to estimate the average cost.

The change in the cost shown to deliver the most recent FORTRAN projects reveals that, due to the overhead of maintaining the reuse libraries, there has been less net improvement in the efficiency of FORTRAN development since adopting the high-reuse process. This suggests that, although FORTRAN is probably more cost effective for building short-lived, single-use software, Ada is preferable for software that is likely to have a longer life through future reuse.

The effect on code size when expressing general software in each language also must be taken into consideration when using statement counts to

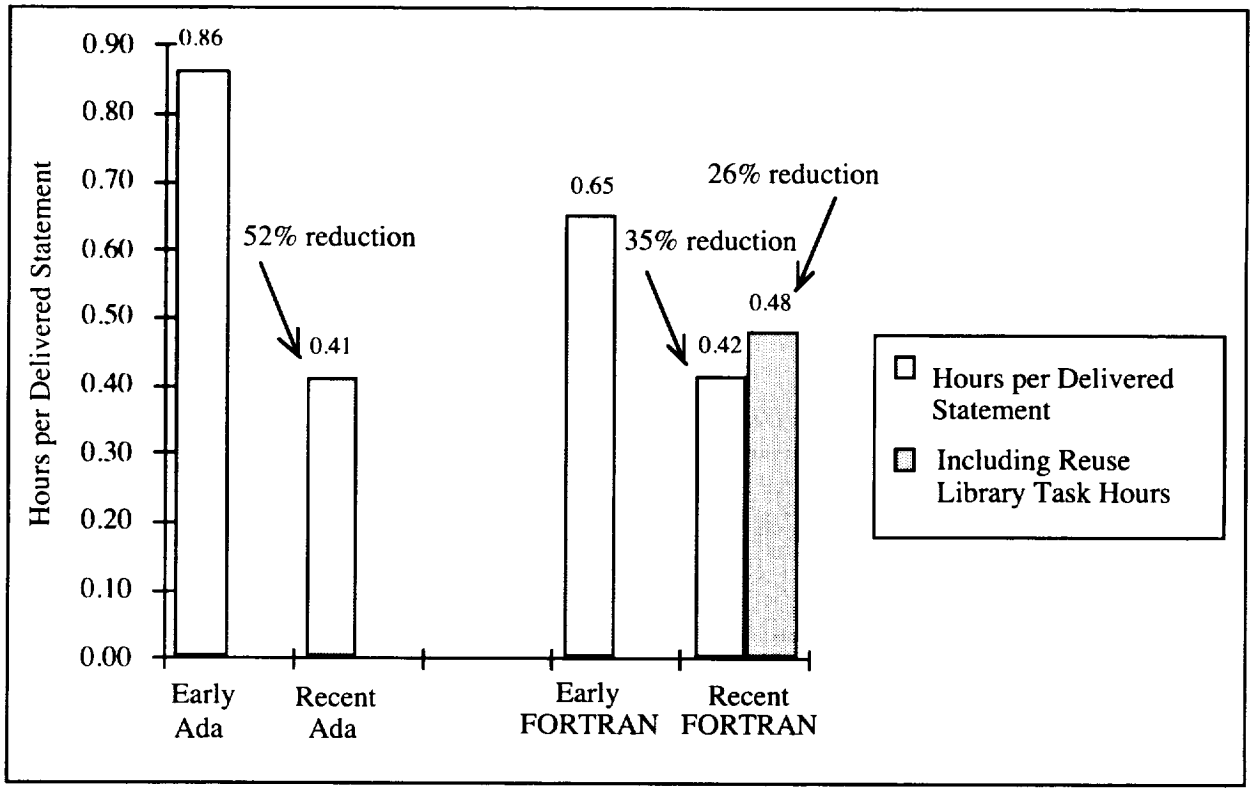


Figure 10. Average Effort to Deliver a Statement

compare productivity. As discussed in section 3.2.3, the generalized portions of the FORTRAN code were larger than the comparable single-purpose solutions. Conversely, the Ada projects that incorporated the reusable generics were somewhat smaller than the earlier similar projects. The net difference between the two languages, which may be as much as 1.8 to 1, means that the effective cost for Ada (based on functionality delivered) is actually lower than that shown above, whereas the effective cost for FORTRAN is actually higher. Adjusting for the Ada size compression factor of .75:1 and the corresponding size expansion factor of 1.25:1 for the generalized parts of the FORTRAN systems (see section 3.2.3) results in a more accurate picture of the change in cost due to high reuse in each language. Figure 11 shows the cost of delivering comparable functionality between the early project set and the recent project set for both Ada and FORTRAN. This indicates that, in terms of functionality, FORTRAN development costs have decreased only slightly, whereas Ada costs have come down by one half due to high reuse.

The current model used in the FDD for estimating the cost of reuse was developed based on empirical data available in 1993.[11] It specifies that development by reuse in FORTRAN costs about 20% of the cost of new code development, but that reuse in Ada costs about 30% of the cost of new code. These figures are the "reuse factors" for each language, which can be multiplied by the new code development costs to estimate the cost of delivering reused software. This model suggests that it costs 50% more to reuse Ada over FORTRAN from the reusing project's point of view.

The findings in this report suggest that the apparent advantage that FORTRAN reuse has over Ada reuse is created by the highly productive black-box verbatim reuse used on FORTRAN projects, which is not available to the Ada projects. The cost of the separate task which offloads the actual expense of the black-box code (i.e., the effort to understand and modify the FORTRAN utility subsystems), is not included in these reuse cost estimates because it is funded separately and available to all FORTRAN AGSS projects. However, because the separately funded cost of maintaining the reusable libraries raises the true cost of the FORTRAN projects in a way that is not reflected by these models, the reuse cost factors are not directly comparable.

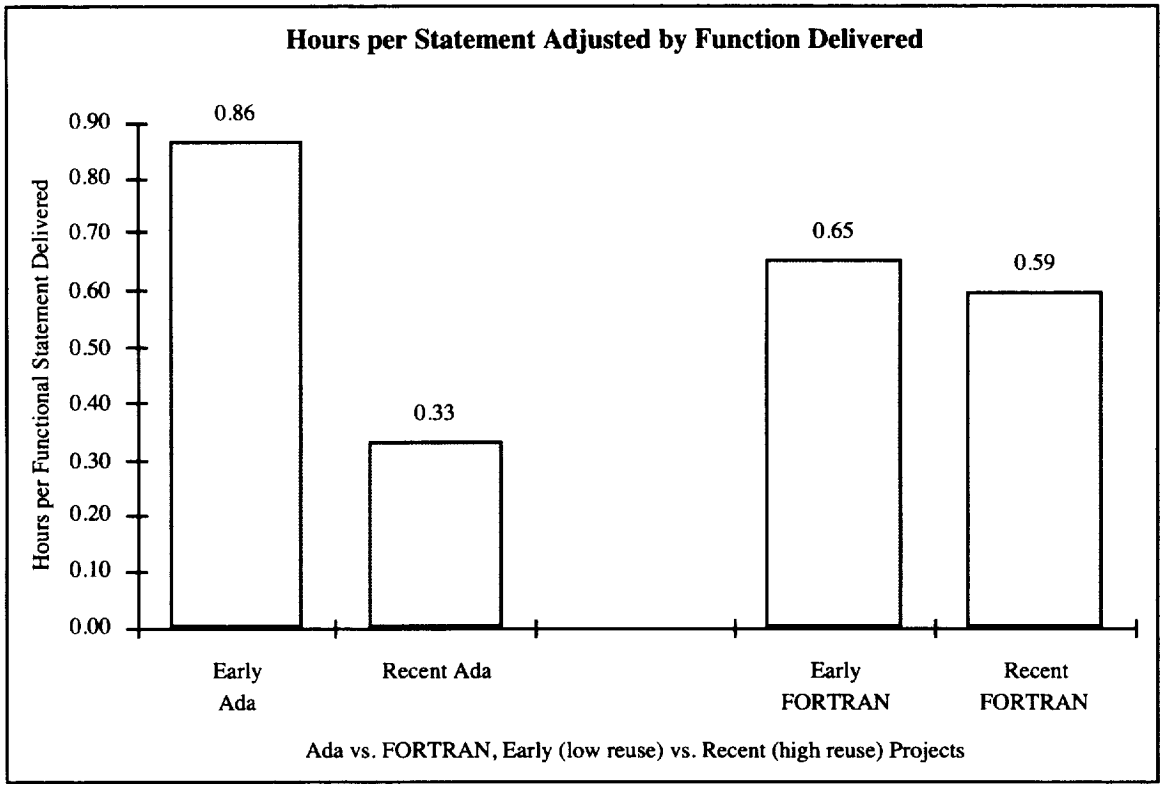


Figure 11. Average Effort to Deliver Similar Functionality

A better way to look at the relative costs of reuse in the two languages is to consider the ratios of productivities between new and reused code in each language, as was done in section 3.2.5. These ratios appear to be nearly identical (except for reuse with modification, where, in the FORTRAN case, a separate team performs the modifications, and productivity rates diverge accordingly), which suggests that similar reuse processes result in similar productivity levels, regardless of language. In fact, it even appears that the per-line productivities are comparable between the languages, which should further simplify future cost models.

3.5 Schedule Compression

In addition to lowering cost, Ada and reuse were also expected to lead to shorter cycle times or project durations. Figure 12 shows that this goal was met not only by the Ada projects but also by the FORTRAN projects. Again, the right-hand bars represent the "recent" projects, or those that achieved high reuse levels. Because this is a schedule comparison, no adjustment is needed to compensate for the MTASS/MSASS effort. The division of labor and reduction in communication that is made possible by having these separate teams, however, is likely to be responsible for shortening the recent FORTRAN development schedules. Because the FORTRAN AGSSs and the Ada telemetry simulators are affected by different external forces, a cross-language comparison of cycle time makes no sense. But a comparison of the early and recent project groups in each language shows improvement.

The software development process did not change immediately with the advent of high reuse, however. As mentioned in the discussion of process, the schedule for the first high-reuse project in each language was more similar to those of the early projects than it was to those of the recent, high-reuse projects. Change in the overall development process occurred only after the EUVE project demonstrated that substantial savings could be achieved through large-scale reuse. When management was able to observe the potential savings from reuse, procedural and scheduling changes were made to allow an expedited development process whenever high reuse was possible. Reuse can permit shortened project schedules, but it is necessary to accommodate this different scenario with an appropriately pared-down process. See section 3.3 for a discussion of these process changes.

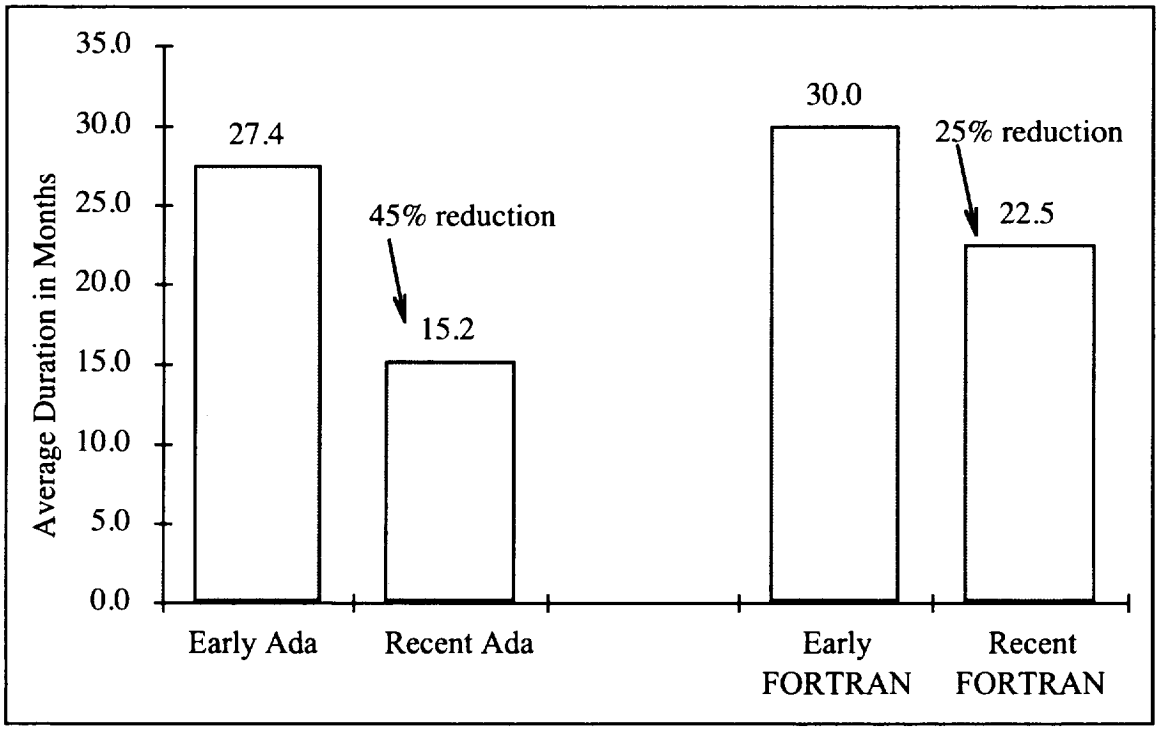


Figure 12. Average Project Duration

3.6 Reliability

The last explicit goal for the planned Ada transition was to increase the quality of the delivered systems. The density of errors discovered during development, which is measured on all FDD projects, was used to represent system quality and reliability (there was insufficient operational data to conduct a reliability analysis). Development-time errors are a useful indication of quality because they reveal the potential for latent undetected errors and indicate spoilage and rework during development.

The number of errors discovered per thousand statements of new and modified software before delivery is shown in Figure 13. The densities shown are based on only the new and modified code (verbatim reused code was not included in the denominator), so these reductions cannot be attributed to reuse. Instead, the reduced error rate is attributed to improvements in the development process that were instituted on all FDD projects during this period. These improvements included the use of object-oriented or encapsulated designs and the use of structured code reading and inspections. The fact that these process improvements were applied to projects in both languages is reflected by the similarity in the error-density reductions observed. The error density reductions were significant at the .01 level for both languages (using a two-tailed Student's t-test).

3.7 Performance

System performance was not an explicitly stated goal for the programs developed in Ada, but it turned out to be a major issue. By 1985, the programmers in the FDD had achieved such proficiency with FORTRAN software design and implementation that even the most complex flight dynamics systems performed adequately without any special attention being paid to performance issues during design. Thus, performance had become an implicit expectation and was not addressed in software requirements, designs, or test plans.

Figure 14 depicts the relative response times of the delivered simulators between 1984 and 1993, where the response time indicates the wall-clock time required to simulate an interval of data (hours responding per hour of data). A smaller response time indicates better performance. The figure reveals that the first Ada simulator performed very poorly

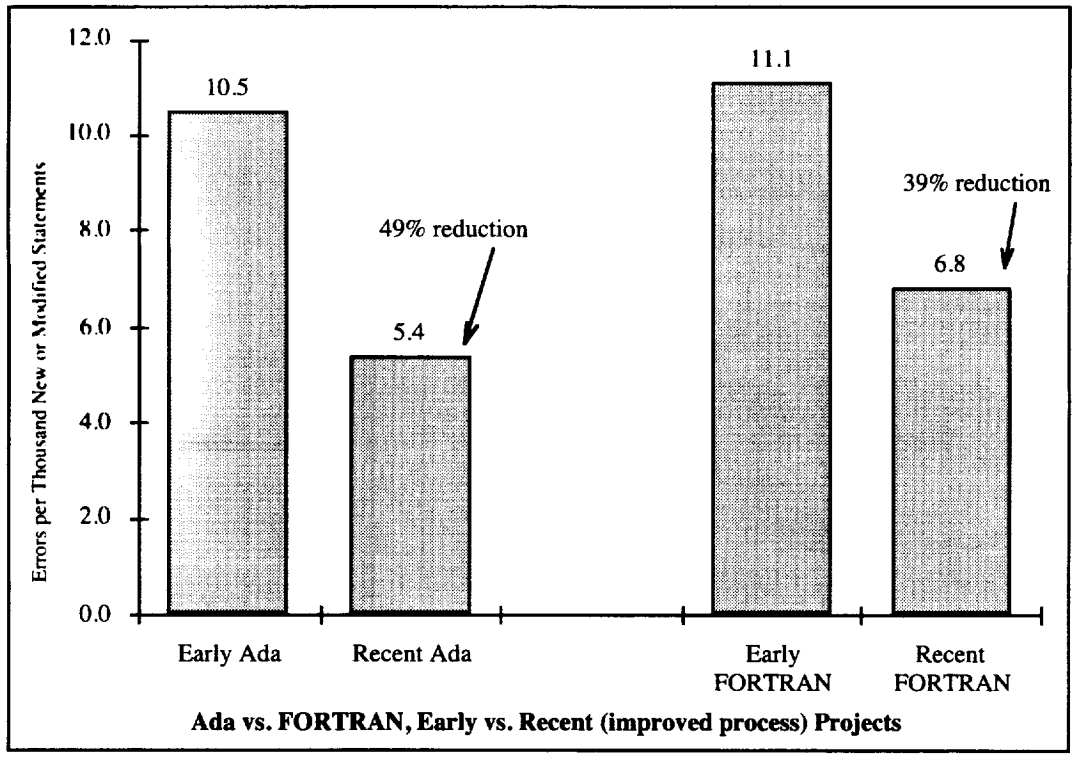


Figure 13. Error Densities on Early and Recent Ada and FORTRAN Projects

compared with predecessor FORTRAN simulators. Because Ada language benchmarks had shown that Ada executed as fast as equivalent FORTRAN programs and because performance was not an explicit goal, developers of the first Ada project paid little attention to performance. Instead they focused on learning the language and developing reusable, object-oriented software. Predictably, as novice users of this fairly complex language, they did not produce an optimum design or implementation. But, their system was delivered for operational use, so the FDD users' first encounter with an Ada system was negative. This impression was accentuated by the fact that, because of scaled-down processing requirements, the FORTRAN simulator delivered immediately before the first Ada simulator was the fastest simulator ever delivered in this environment.

In retrospect, it appears that attempts to maximize use of OOD while lacking extensive experience with the technology probably contributed more to the initial poor performance than did Ada. This points to a basic flaw in the approach taken to the evaluation of this new technology: Conflicting goals had been established for Ada by combining its study with the use of OOD. It was then difficult to separate the effects of the language from the effects of OOD techniques, resulting in the language being faulted for the run-time overhead caused by data access procedures and by multiple layers of abstraction.

In addition to the overhead from OOD, the 1990 Ada performance study[8] revealed that some of the coding techniques practiced in FORTRAN to achieve high efficiency actually worked against efficiency in Ada, and that some of the data structures around which the designs were built were handled very inefficiently by the DEC Ada compiler. The study resulted in a set of Ada efficiency guidelines[9] for both design and code, which are now being followed for all new Ada systems. Interestingly, to comply with those guidelines, the last two Ada projects in Figure 14 had to forgo a certain amount of reuse (compare with Figure 5). (The slower POWITS simulator was completed before these guidelines were available and it also had considerably more complex processing requirements as well as other complications.)

As shown in Figure 14, a typical Ada simulator now performs better than most of the earlier FORTRAN simulators. First impressions are very important, though, and some FDD programmers and users still hold the perception that Ada has performance

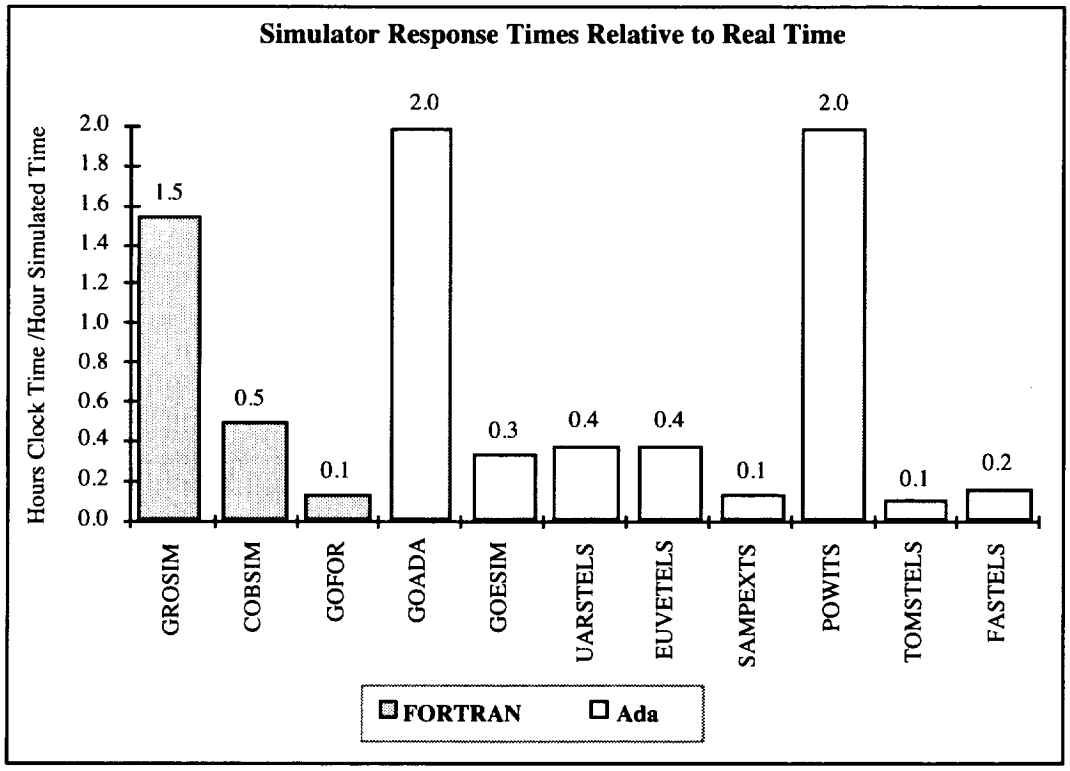


Figure 14. Performance Times of Ada and FORTRAN Simulators

problems and that systems demanding high performance should not be implemented in Ada. Recent impressions have been more favorable toward Ada, however. These impressions are discussed in detail in section 4. In fact, subjective data collected about Ada simulator performance (analyzed in section 4) reveal that those with recent Ada experience have no complaints about performance. Another indication that performance is no longer an issue is that performance benchmarks are no longer run for the Ada products, although current performance requirements are more demanding.

3.8 Summary of the Comparisons

The quantitative data gathered over the past 10 years show clear improvements attributable to the use of the Ada language in all of the initially specified goal areas. Many of these improvements were directly related to a considerable increase in reuse. Although the FORTRAN systems gave the impression of comparable results over the same period, the cost savings from reuse were considerably lower in FORTRAN when the added cost of maintaining the reuse libraries was factored into the net delivery costs. Further, the expansion of statements per unit of functionality needed in the high-reuse FORTRAN projects further offset the apparent gains in productivity as measured simply by the size of the delivered product.

With respect to improvements in schedule and error density, Ada and FORTRAN performed comparably. The initial disappointments with Ada with respect to performance were subsequently eliminated through analysis and by the adoption of performance-sensitive Ada design guidelines.

Therefore, the study found no quantitative evidence to indicate that the Ada language can not be used successfully for all FDD projects. In fact, the data show every indication that Ada is a good choice for increased usage on more FDD systems, particularly larger, longer-lived, highly reusable systems.

Section 4. Qualitative Analysis

Despite the promising quantitative results that accrued from the use of Ada, the adoption of Ada at the FDD was slower, more difficult, and less widespread than expected. As is often the case with technology infusion, several external and internal subjective factors impeded the FDD's transition to Ada. Factors such as the limited availability of Ada compilers and tools, negative feedback from the users of the developed systems, and an adverse and vehement minority opinion within the software development organization all had detrimental effects on the adoption of Ada. This section discusses these factors and their impacts on the goal of transitioning to Ada.

4.1 Vendor Tools and Support

Finding adequate vendor tools to support Ada development in the FDD was a major obstacle. In 1985, when the FDD began its work with Ada, most computer vendors were either actively developing Ada compilers and development environments or had announced plans to do so. The FDD believed that, within a few years, vendor tools would be widely available for Ada. But, consistently usable Ada development environments and reliable Ada compilers never became available across the platforms used at the FDD to develop and execute software systems.

With only one small exception, all the Ada projects at the FDD were developed using DEC Ada on VAX minicomputers. The DEC/Ada products available on the VAX platforms were rated by FDD developers as being sufficient to enable viable Ada development. However, 80% of the software developed in the FDD must execute on the standard operational environment, which is an IBM mainframe. Traditionally, FDD systems have been developed on their target platforms because this simplifies testing and deployment. Unfortunately, an adequate Ada development environment was never found for the IBM mainframe.

In search of a solution to mainframe development, the FDD conducted three studies between 1989 and 1992, all of which declared the IBM mainframe environments unfit for Ada software development or deployment. In 1989, the FDD evaluated three compilers[18] and selected one for purchase and further study. Somewhat discouraged by this study, which rated the best compiler as having only marginal performance for flight dynamics computations and no development tools, the FDD investigated an alternative approach.

Because Ada was touted to be highly portable, the FDD conducted an alternative portability study to determine whether systems could be developed in Ada on the VAX and then transported to the mainframe for operational use. This study (discussed in section 2.4.3) ported one of the existing operational Ada simulators from the VAX to the IBM mainframe using the Alsys IBM Ada compiler, version 3.6. The study found that relatively few software changes were required and that the resulting system performed adequately on the mainframe, but that rehosting was extremely difficult because of compiler problems and the lack of diagnostic tools and library management tools. Although rehosting the system required only a small amount of effort, it took nearly as much calendar time as was needed to develop the system from scratch, due to the problems encountered.

An alternate approach would have been for the FDD to purchase the Rational Ada Environment, which would have allowed development of Ada components specifically tailored for IBM mainframe compilation. This cross-targeting strategy, developed by Rational to solve many of the problems associated with delivering Ada software on mainframes and other platforms with inadequate Ada environments, would have involved purchasing additional hardware as well as software tools and licenses. Given that the VAX provided a viable Ada development environment, the FDD did not seriously consider converting to the Rational development environment. Additionally, the cost to use Rational in 1991 would have come to about $35K in hardware and software per seat, an amount which the FDD deemed excessive.

In the fall of 1992, the FDD again conducted a compiler evaluation on what were supposed to be greatly improved products. This study[19] used the ported simulator as one of its benchmarks and ended up selecting a different compiler than the earlier study. Although the chosen compiler performed better than other candidates and was accompanied by

a modest tool set, the study warned against using it to develop real-time or large-scale FDD systems because of its inefficient compiling and binding performance, immature error handling, and poor performance of file input/output. Only in late 1993 did the FDD achieve some limited success developing a small utility in Ada (the FAST General Torquer Command Utility*) on an IBM RS-6000 workstation and then porting and deploying the software on the mainframe. This approach to developing Ada systems for mainframe operation is the first to show any real promise. In addition, this has been the only instance of Ada software development on a workstation platform. While the FDD had hoped to begin earlier investigating the appropriateness of using workstations to develop and execute Ada systems, the cost of suitable software development environments on workstation platforms has been prohibitive.

In addition to its disappointment with the mainframe development environments, the FDD also experienced only qualified success using Ada to develop an embedded system. The FDD's R&D effort to develop an embedded application on a Texas Instruments 1750A machine using the TARTAN Ada compiler led to interface problems between the hardware and software. Ultimately, the lack of diagnostic tools contributed to insoluble problems that resulted in an end product with reduced capability (discussed in detail in section 2.3.1). This experience added to the general feeling among FDD developers that the level of vendor support available was unsatisfactory for viable Ada development.

4.2 Ada Perspectives Within the FDD

Technology transfer of any software engineering technology involves people: users, software developers, and managers. The introduction and usage of Ada within the FDD sparked much controversy. The independent assessment team sought to determine the impact of the Ada technology on the people of the FDD and to understand the degree to which the attitudes of the various groups impeded or facilitated the infusion of Ada in the FDD. This section presents the key findings from interviews and surveys conducted during the independent assessment.

4.2.1 User Perspective

As mentioned in section 2, the SEL conducted a performance study in 1990 largely because feedback from the user community indicated that the Ada systems were not as fast as their FORTRAN predecessors, and were therefore unacceptable. A survey of 18 users taken in late 1991 showed that performance ideals varied considerably among the users of the satellite simulators. Performance goals ranging from one-quarter real time to 15 times real time were cited by the users, with the most frequently cited performance goal being at least real time (where the simulation of 1 hour of data takes 1 hour of wall-clock time). Most of the *recent* Ada simulators have exceeded this goal; however, when the 1991 survey was taken, only SAMPEXTS, with a simulation speed of about 3:1, clearly exceeded the 1:1benchmark. Although the users realized that the complexity of a simulation and the speed and availability of the hardware also affected performance, most blamed the simulators' poor performance on the Ada language itself.

More recent feedback from the users of the Ada telemetry simulators has been entirely favorable. Performance results ranging from 5 times real time to as much as 15 times real time are now being reported. In fact, performance is no longer even considered an issue to the users of the Ada simulators. One indication of this is that no performance benchmarks have been run or requested for quite some time. Likewise, interviews with the AGSS testers and mission-support users, who use the simulators for prelaunch testing of the AGSSs and, occasionally, for testing of emergency repairs during mission support, did not yield any complaints about the performance of recent Ada simulators. The users have also stated that changes to the Ada simulators have been easy to specify and obtain. They routinely participate in the analysis of changes recommended to accommodate new requirements.

This current situation with respect to the usability and performance of the Ada simulators is in stark contrast with the situation reported even as recently as 1992. In fact, a major motivation for conducting the independent assessment was to determine the future course of action to address the problems encountered using Ada at the FDD. Apparently, the efforts of the

*This FDD Ada product is not included in the project data analyzed for this report. It is a unique entity in this environment in that it is an AGSS subsystem written in Ada; therefore, it did not fit into any of the defined project sets.

software developers to focus on and improve performance of the TOMSTELS and FASTELS simulators in particular was well worth the sacrifice in reuse.

4.2.2 Developers' Perspective

During the past 2 years, the independent assessment team conducted two surveys to gather insight into the perspective of the software development staff. The first survey addressed those developers who had direct exposure to Ada, those who either used Ada on the job or attended Ada training, to measure their attitude about the language. The second survey, which addressed the total FDD software development population, measured how each respondent felt about the future of Ada in the FDD. Both provided insight into the overall impact that the introduction of the Ada technology has had on the people in this organization. Key findings from the analysis of these survey results are presented in this section. The survey forms are included in Appendix C.

The first survey gathered information from 35 FDD developers who had been trained in or had developed systems in Ada. Developers were asked which language they would choose for the next simulator project, which language they would choose for the next ground support system, and why. Figure 15 shows the sum of their responses.

Most agreed that Ada should be used for the next simulator, but that FORTRAN should be used for the next ground support system, citing the availability of reusable components and architectures as the deciding factors. But, significant minorities in each case recommended use of the language not customarily used for each type of application. The 23% who preferred to use FORTRAN instead of Ada for simulators cited the complexity of the Ada language and poor performance as reasons to abandon Ada, while the 30% who preferred to use Ada instead of FORTRAN to build the next ground support system felt that Ada was a better language for building larger systems. Interestingly, several of the developers did not care one way or the other about which language they used for software development, with two developers specifically commenting that "Ada is just another language."

Nearly all the developers exposed to Ada pointed out that adequate tools are essential for efficient and accurate Ada development, whereas FORTRAN development can be accomplished with little or no external tool support. In particular, they cited the need to have tools to help them with the Ada compilation dependencies that allow Ada's sophisticated error checking during compilation. Interestingly, those who had training followed by on-the-job experience responded positively about Ada, whereas those who had training and no hands-on

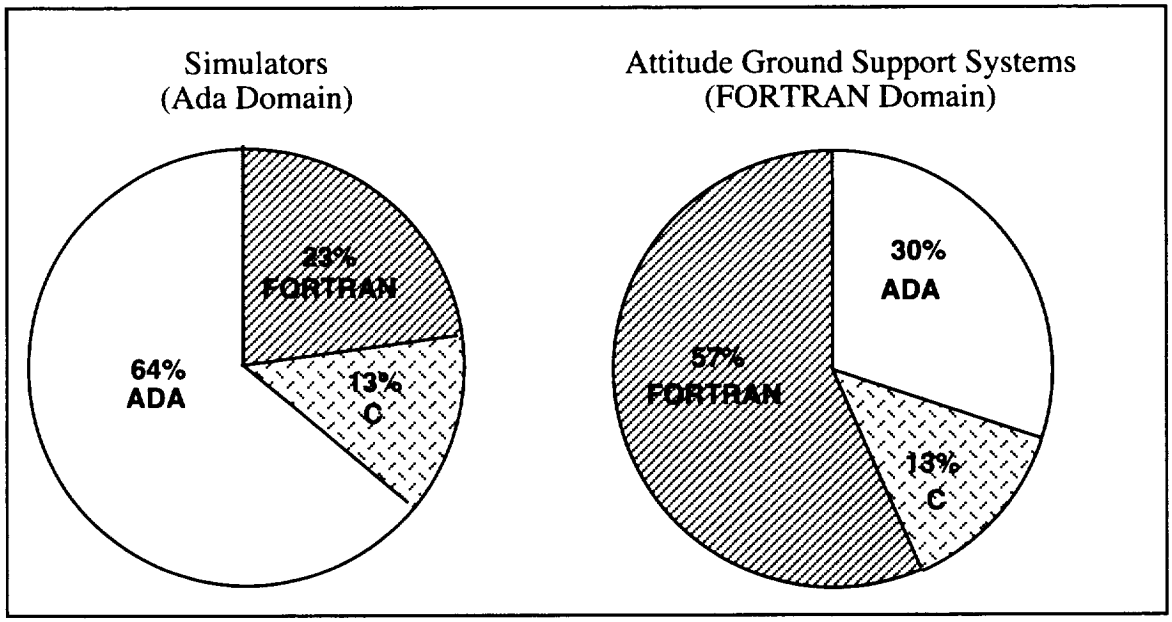


Figure 15. Language Preference for FDD Systems

work experience using Ada had a consistently negative opinion of the language. This indicates that the language is hard (complex) to learn, but that, with day-to-day experience, one becomes proficient quickly and experiences the benefits of the language.

As of March 1994, only 25–30% of the development community had been directly exposed to Ada. However, it was clear from interviews and discussions with FDD personnel that there had been a broader impact on the organization as a whole. Two significant minority groups had emerged who were strongly opinionated about language use, one in favor of Ada and the other opposed. Both groups had been fairly vocal and forthcoming with their views throughout the transition to Ada. The second survey was designed to capture the views of the developer community as a whole and to look for a possible effect that these vocal minorities may have had on the remaining group of developers who had not yet been trained in or exposed to Ada.

The second, broader survey (see Appendix C) collected responses to questions about basic background information as well as opinions about the use of Ada at the FDD. Background information included job category, FDD experience, computer language experience, and Ada exposure. Ada opinion questions included whether the use of Ada was appropriate at the FDD, whether Ada should be restricted, whether its use should be increased, and whether its use should be decreased. The survey team collected 103 responses from developers (including maintainers and testers), 15 responses from managers, and 7 responses from SEL researchers and others. In order to ensure candid responses, respondents were given the option to return the surveys anonymously.

Approximately half of the developers answered "don't know" or "don't care" to all four Ada opinion questions. Of the half who expressed opinions, a clear majority was positive about the appropriateness of Ada at the FDD. Most felt that the level of usage should remain roughly constant, neither expanding nor reducing the amount or type of application software developed in Ada. Table 8 presents the responses for those who expressed an opinion. The balance of this section summarizes the views of the developers surveyed.

The backgrounds of those with the strongest negative opinions about Ada provided some insight into probable causality. Source of Ada information and knowledge appears to be a key contributor. Of those with the most negative responses, only 1 in 8 had on-the-job experience with Ada. The others had only taken an Ada class or self-studied it, or had no real exposure to Ada at all. (Among those expressing opinions in general, fully one-third had Ada work experience, confirming that work experience improves one's opinion of the language.) Responses from those developers who received their information about Ada only from others at the FDD indicated that negative opinions about Ada were more likely than positive ones to influence those with no formal Ada exposure.

The survey also revealed a slight negative effect from in-house Ada language training when it was not followed by Ada work experience. This supports anecdotal evidence that the in-house training given at the FDD was detrimental to the typical developer's opinion of the language, while further confirming the positive effects of Ada work experience. Subsequent investigation revealed that responses varied depending on the specific class and instructor who conducted the Ada training. The classes conducted by a trainer from an outside organization were generally better received than those conducted by in-house Ada experts.

Table 8. Ada Survey Responses for Developers Expressing Opinions

Ada Opinion Questions	All Developers		Developers Without Ada Experience	
	Yes	No	Yes	No
Is Ada appropriate in the FDD?	79%	21%	50%	50%
Should Ada use be restricted?	44%	56%	67%	33%
Should Ada use be increased?	35%	64%	0%	100%
Should Ada use be decreased?	37%	63%	100%	0%

The length of time spent at the FDD, the number of years of FORTRAN experience, and the number of computer languages known had no effect on a developer's opinion of Ada. However, a higher than average number of recommendations to decrease the use of Ada came from the customer organization as compared with the FDD contractor organization. To obtain a more complete picture of the range and distribution of Ada opinion, including the distribution by organization, the responses to the four questions were converted to composite scores. Positive values were assigned to positive Ada opinions and negative values to negative opinions. Zero values were assigned to "don't know" or "don't care" responses.*

Figure 16 shows a frequency distribution of the composite scores. The tallest bar, at the neutral score of zero has been truncated to clarify the shape elsewhere in the histogram. The tendency for frequencies to diminish outward from the center is interrupted by "bumps" in both tails. These bumps in the curve at both the extreme positive and the extreme negative scores reflect the strongly opinionated and polarized minorities on both sides of the Ada issue. Opinions expressed by the bulk of the respondents, though, fell squarely in the middle, indicating a vast majority having no bias whatsoever.

The contractor organization expressed a more positive overall opinion of Ada than the customer organization. Contractors believed that exposure to and experience with new technologies would make them more marketable and would lead to better future career opportunities. The marketability of Ada developers was confirmed in 1989 when the contractor organization lost several of its most experienced Ada developers after the initial Ada projects were completed. For various reasons, often purely economic ones, several developers chose career moves away from the FDD at a critical time in the Ada transition. Although some of the most knowledgeable Ada developers remained in the FDD, this migration removed a core of Ada experience and opened the door for many new developers to gain Ada experience. Had this exodus not occurred, the subsequent Ada projects would probably have proceeded more smoothly, resulting in a more positive attitude towards Ada among all FDD developers. Nevertheless, the remaining developers in the contractor organization learned first-hand of the opportunities available to their colleagues with Ada experience. Now, in the mid-1990s, C and C^{++} seem to have replaced Ada as the languages that developers feel will make them more marketable.

The written comments on the survey forms expressed additional observations, perceptions, and points of view about Ada. The most prevalent theme among these comments was the need for adequate tool and vendor support when committing to Ada. Specific references were made to the incompatibility of Ada and the IBM mainframe architecture as well as to the need for reliable vendor support. The lack of readily available packages for interfacing Ada with software toolboxes and other languages was also cited as detrimental.

Next to inadequate tool support, the most commonly mentioned point about Ada was the difficulty experienced in learning and using the language properly. Five developers said either that Ada was hard to learn or that other languages, such as C, were easier to use. Additional specific disappointments included difficulties with Ada input and output and the complexity of doing true OOD with Ada.

Not unexpectedly, the written comments tended to parallel the Ada opinion scores obtained from the other questions in the survey. The polarity of opinion present at the FDD can be seen here, because the strongest opinions, both negative and positive, were usually held by those at the extreme ends of the opinion score distribution. Overall, there were three unconditional endorsements and six qualified endorsements of the use of Ada in the FDD. On the other hand, five respondents wrote completely negative comments about Ada and another seven were generally pessimistic or skeptical about Ada.

4.2.3 Management Perspective

Fifteen managers provided responses to the second Ada opinion survey. Among the management subset, the average tenure at the FDD was greater than 13 years, as opposed to less than 6 years average

*Weightings were applied to reflect the strength of opinion indicated by the response. The weighting scheme was tuned slightly in order to normalize the sum of all developer opinion scores to near-zero (i.e., so distribution of scores was balanced on either side of zero). The weights for each response did not appear on the original survey questionnaire. Because the questions were designed to be particularly revealing of negative opinions, the lowest possible score is -7 while the highest possible score is +5. Independent of the high frequency of zero scores due to "don't care" or "don't know" responses, the distribution appears to be roughly balanced (the average of all non-zero scores was -0.02, or nearly zero).

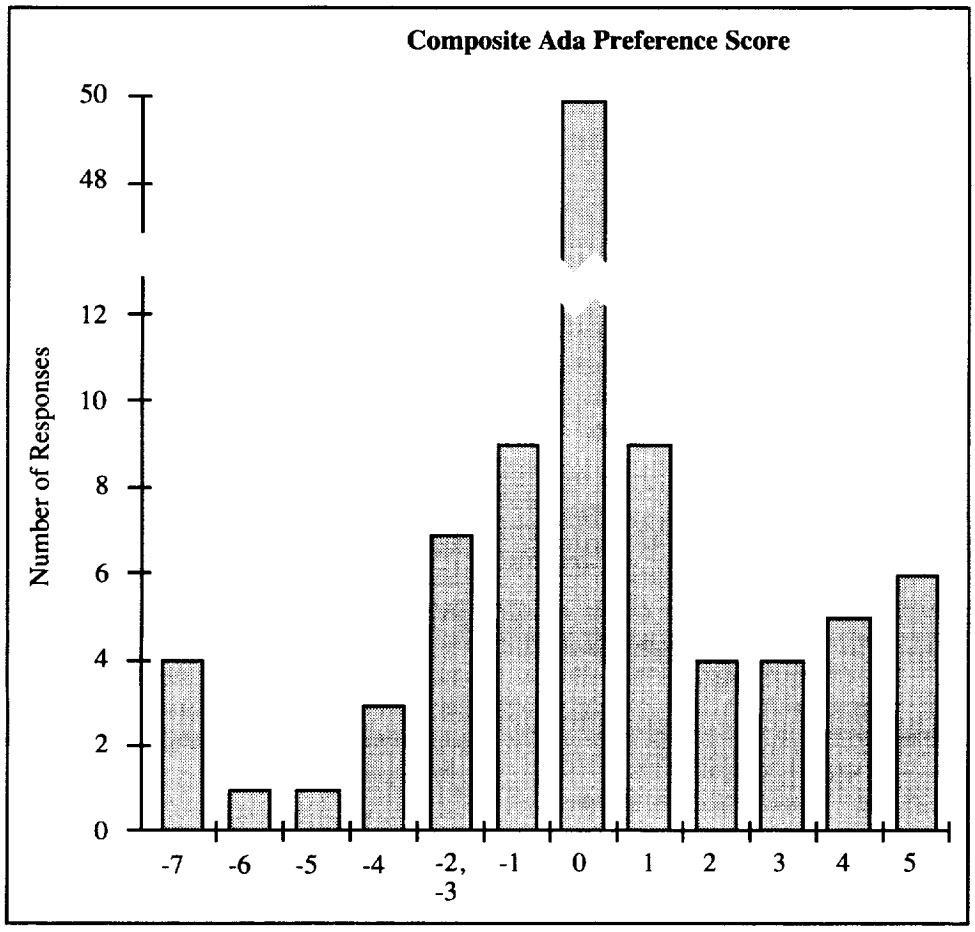


Figure 16. Distribution of Developers' Ada Preference Scores

FDD experience among the developer subset. However, the average manager had facility with only two languages whereas the average developer knew more than four. The average manager had 15 years experience in FORTRAN whereas the average developer had about 9 years of FORTRAN experience. Fewer managers had computer science or physics backgrounds but slightly more had mathematics backgrounds as compared with developers.

About half of the managers (8 of the 15) had Ada exposure but only one had actual on-the-job experience using Ada (one other had managed an Ada project). Classes or seminars in Ada constituted the only exposure to the language among the other six. The half with no Ada exposure (7 of 15) obtained their information from others both within and external to the FDD; only one cited additional sources for his knowledge of Ada, including literature and conferences.

The average management composite score was slightly more positive towards Ada than the average developer score. In general, manager opinions reflected those of their staff. In fact, the division between the customer and contractor organizations was the only clear correlate to the Ada opinion score, with the 5 managers from the customer organization averaging to a net negative opinion and the 10 managers from the contractor organization averaging to a net positive opinion about Ada.

Interestingly, the biggest difference between developer and management opinion came from the substantially greater percentage of managers who favored restricting the use of Ada as compared with developers. This appears to be a sign of caution among managers. When compared with developers, managers did not express any greater or lesser interest in either expanding or reducing the use of Ada, however they did more often wish to avoid the unrestricted use of the language.

4.3 Net Result

Figure 17 depicts the growth of Ada software being delivered each year during the Ada study period. A sharp decline in the amount of development occurred in late 1990. It was at this point that the FDD had planned to begin developing parts of the larger ground support systems in Ada on the mainframes. However, the results of the early Ada compiler evaluation and the portability studies made it clear that developing on the mainframes, or even developing elsewhere and porting to them, was not feasible. Thus, the growth of new Ada development stalled at this point.

At this same point in time, the FDD's simulation requirements changed, reducing the number of simulators needed to support each spacecraft mission from two to just one. This change resulted in a further reduction in the amount of software slated for development in Ada. The net result was a substantial reduction, instead of the envisioned increase, in the rate of Ada software delivery. The drop in Ada development is even more dramatic when the amount of reused software is eliminated from the totals and only the investment in new Ada code is considered. The flatter, dashed line below the curve for cumulative delivered size in Figure 17 removes the effects of reuse by showing only the number of new and modified lines that were delivered.

The unavailability of an adequate Ada development environment on the IBM mainframe was clearly a significant stumbling block for the FDD in its transition to Ada. Had the FDD been able to expand Ada development into the mainframe environment as originally planned, much of the operational software that now exists in FORTRAN would have been written in Ada. Much more of the staff would have gained hands-on work experience in Ada, which, based on the data presented in section 4.2.2, would have led to a more positive reaction to the language.

If the FDD were to continue to use mainframes as its principal operational environment, there would be no straightforward way to fully transition to Ada. However, the FDD has committed to and has begun transitioning to open systems for operational support. In the future, software will be developed and deployed on workstations in a networked environment. Thus, a full transition to Ada will depend on the viability of using it for workstation development on a larger scale.

A recent FDD internal study found that Ada development environments are very expensive compared with development environments for other languages that support OO development. The typical cost for an adequate Ada development environment for a single workstation seat ranges from $8.5K to $17K, depending on the quality and completeness of the tool suite. Conversely, a comparable development environment for C or C++ ranges from $2K to $3K per workstation seat. Thus, the high cost of workstation development environments now poses the most serious risk to the future use of Ada in the FDD.

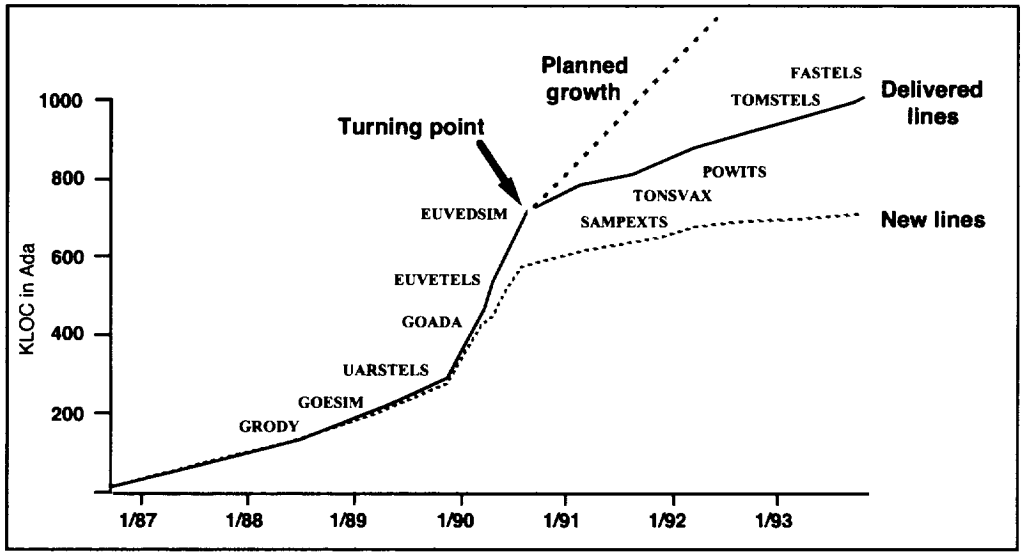


Figure 17. Growth of FDD Ada Software

Section 5. Conclusions and Recommendations

Overall, the FDD benefited greatly from its exposure to and work with Ada. Although, nearly 10 years after Ada's introduction, the FDD uses it to develop only 15–20% of its software, many of the concepts and disciplined software engineering practices associated with Ada have been adopted in the development of all new systems, no matter what language is used. By using OO techniques, such as domain analysis, data abstraction, and information hiding, the FDD has increased its reuse of software by 300%. This in turn has led to reduced mission cost and cycle time for FDD products. Thus, the FDD achieved its original goal of reducing cost and cycle time by maximizing reuse via the introduction and use of the Ada language and OOD.

Although the SEL's assessment of this technology has shown it to be beneficial, it is unlikely that the FDD will fully transition to Ada as its language of choice. The lack of mainframe development environments and the high cost of viable Ada software development environments for workstations continue to be a barrier against using Ada to develop the bulk of the FDD's systems. Up until now there has been no driving reason to change languages. However, the results documented here show good reason to move away from FORTRAN. As it moves to a distributed workstation hardware environment, the FDD has the opportunity to select a new, cost effective language(s) for its future. Weighing the tradeoffs between short-term costs, such as software development environments for workstations, against software development process and product issues and the long-term costs of software maintenance, the FDD is likely to find Ada a good choice.

The key findings and technology transfer lessons learned from this research and analysis are summarized below. Recommendations are made regarding the future use of Ada in the FDD.

Key Findings

- *Use of Ada and OOD in the FDD resulted in:*
  - *Increased software reuse by 300%*
  - *Reduced system cost by 40%*
  - *Shortened cycle time by 25%*
  - *Reduced error rates by 62%*

By 1990, projects using Ada and OOD were experiencing measured improvement. When compared with the SEL baseline that existed when the Ada assessment began (1985), projects using Ada showed improvement across the board in cost, schedule, and quality as a result of achieving unusually high levels of reuse.

- *The experimentation with Ada and OOD served as a catalyst for many of the improvements seen in the FORTRAN systems during the same period.*

In 1985, Ada was arguably more than just another programming language. However, by exposing the organization to the concepts of information hiding, modularity, and packaging for reuse, that which was "more than a language" was adopted, to the extent possible, by the FORTRAN developers as well as by the Ada developers. Anecdotal evidence supports the theory that Ada served to catalyze several language-independent advances in the ways in which software is structured and developed across the organization, and that these benefits have been institutionalized by process improvements. The exposure of many managers and application experts to object-oriented design via Ada projects served to open their minds to new ideas on other projects. FORTRAN AGSS designers met significantly less resistance to using object-oriented concepts to redesign the well-established, well-understood standard architecture for AGSSs than had been typical when design alternatives were proposed previously.

- *FORTRAN systems applying object-oriented concepts also showed significant improvement in reuse. Like the Ada projects, higher reuse led to reduced cycle times and lower error rates on the FORTRAN projects. However, they did not experience similar cost savings; use of Ada resulted in greater cost reductions for systems with roughly comparable levels of reuse.*

The FORTRAN systems also showed improvements in schedule duration and quality attributable to increased levels of reuse when compared with the 1985 baseline. However, the cost reduction was not nearly as significant as with the Ada systems. This was largely due to the effort required to maintain the reusable software. Whereas the use of Ada generics

allowed project personnel to reuse code through parameterized instantiation rather than repeated modification, the FORTRAN systems required a separate maintenance team to enhance the reusable components (add new capabilities). Although the separate maintenance team could make the modifications as efficiently as possible (due to familiarity with the code) and the cost of reusing the code from the projects' point of view was virtually nothing, the additional cost of supporting a separate maintenance team nearly negated the savings.

- *Use of Ada resulted in smaller systems to perform more functionality; while generalization increased the size of the FORTRAN systems.*

The use of Ada generics to implement a generalized architecture in the UARSTELS simulator resulted in a system that was 17% smaller than its predecessor (GOESIM) and performed 10% more functionality. Conversely, generalized FORTRAN subsystems are 10–40% larger than earlier single-mission versions. Also, over time, the generalized FORTRAN components have grown as they are enhanced to support new missions, while the size of the generalized Ada components has remained fairly constant.

- *Lack of viable Ada development environments on the FDD's primary development platform severely hampered the transition to Ada.*

When the FDD began using Ada, the availability of vendor tools was of little concern. DoD's mandate that all of its systems be developed in Ada was expected to provide a substantial market for Ada compilers and tools. However, in reality, DoD developed far fewer systems in Ada than expected. This decreased the demand, and vendors lost their incentive to supply Ada support tools. When it became apparent that no vendor planned to provide a full Ada development environment for the IBM mainframe, the FDD had limited options. Because the FDD had just installed a new IBM mainframe, it could not change hardware for at least 5 years. It had neither the money nor the clout (size) that a large company or government agency might have had to offer vendors the incentive to build an Ada environment for the IBM mainframe. Another option available at the time, the Rational development environment, which other IBM-mainframe-based organizations were using, was prohibitively expensive for the FDD.

Thus, in 1990, when the FDD was ready to expand to full use of Ada, it could not. This essentially stalled the FDD's transition to Ada. Although simulators continued to be built in Ada and a small group of people continued to develop plans and approaches for building reusable building blocks and architectures in Ada that would be used to construct systems on workstations in the future, much of the workforce continued to be untouched by the technology. This standstill allowed other languages (such as C, C++) to make advances as viable alternatives to Ada, and allowed opponents of the technology within the workforce to raise doubts about Ada among those who had never been directly exposed to the technology.

- *The high cost of Ada development environments on workstations may deter future use of Ada as the FDD transitions to open systems.*

Today, as the FDD prepares to transition from the mainframe environment to open systems and software development on workstations, the organization is faced with a large investment for new hardware and support software. Ada development environments (compilers and the necessary software development tools) cost significantly more (3–8 times more per seat) than development environments for other languages that are commonly used with OOD, such as C++. This poses a financial barrier against the FDD's future use of Ada that should be weighed against the potential savings of building and maintaining systems using Ada.

- *The introduction of Ada sparked much controversy within the FDD. At this time, most of the FDD workforce is lukewarm toward using Ada, with two vocal minorities for and against its continued use. However, most personnel support the use of object-oriented techniques.*

A definite negative attitude toward Ada exists among a small percentage of developers and managers in the FDD who have no direct working experience with Ada. In addition, two small, but vocal groups of people have demonstrated a very strong bias for and against Ada, respectively. Both of these groups appear to have contributed to the negative bias among the general population: the proponents by overselling the technology and the opponents by negative campaigning. Interestingly, there does not appear to be a corresponding bias against OOD.

Nearly all of the people believe that OO techniques are beneficial and look for ways to apply them, no matter what language they are using.

Technology Transfer Lessons Learned

- *Technology insertion takes a long time, especially when several technologies are combined or when the technology affects the full development life cycle and requires a significant amount of retraining.*

It took approximately 5 years for the FDD to transition to regular routine use of Ada for a particular class of systems. It took nearly 2 years longer to understand the process differences well enough to produce a standard process for Ada projects.

- *Parallel development experiments are an effective way of minimizing the risk of a major new technology to the organization; however, the project using the new technology must be tightly managed to maximize value and minimize negative effects.*

Use of the GRODY parallel development experiment to introduce Ada and OOD to the FDD had both positive and negative effects on the technology infusion process. On the positive side, it eliminated the risk to operational software, thus allowing free and complete exploration of the technology. However, loose management of the experimental project led to inflated functionality and nonadherence to schedules. Because of the inflated functionality, direct comparisons of size and error rates were not possible; and the lack of adherence to deadlines made it hard to compare costs and life-cycle schedules. These factors compromised the integrity of the experiment and contributed to the perception within the FDD that Ada is a "sand-box" (or experimental) technology.

- *First impressions are very important; be careful to understand and set realistic expectations regarding the new technology for everyone affected.*

First impressions caused many problems during the FDD's experience with Ada. Because the developers did not anticipate the impact of the new language and design decisions on system performance, they did not focus on performance requirements during the development of the early systems. Unprepared users were very disappointed in the performance of the early systems and blamed the technology rather than the way in which it had been applied. Today, it is hard to find a dissatisfied user, but it took a lot of effort to overcome the initial impression that Ada was "too slow."

- *Project personnel will focus on and meet the goals set for them at the expense of those not explicitly stated. Be careful to consider all aspects of the new technology when setting goals for pilot projects, and clearly state all goals and their relative priority.*

Each one of the experiments and pilot projects met the goals set for them. However, projects often encountered problems in areas where they sacrificed or overlooked something because of their narrow focus on their primary goal. For example, GRODY personnel explored the new features of the language without paying any attention to system performance. And even after GRODY's poor performance was known, the GOADA and EUVEDSIM teams opted to reuse inefficient code because high reuse was their goal. The GOESIM team sacrificed the use of new Ada features and OO concepts to guarantee delivery on schedule and within budget.

- *New technology advocates are essential to initiate and sustain the technology transfer process. However, if they are not sensitive to the needs and concerns of the organization and its developers, they will impede rather than facilitate the process.*

The FDD had a few respected technology experts who were very knowledgeable about Ada and OOD and who were enthusiastic proponents of the language. Following their lead, the FDD vigorously pursued Ada and OOD and tried many new ideas that moved the technology's application forward in both industry and the FDD. These technology experts or advocates were expected to assist people who were learning and using the technologies for the first time. However, in some cases, the advocates' zeal for Ada and lack of real project experience made them less sensitive to the concerns of the people who needed to use the new language on real projects. Consequently, they provided help with technical problems, but did not acknowledge and constructively discuss others' frustrations with applying Ada. Gradually people became disillusioned with the technology advocates and stopped going to them for help and, in some cases, began to actively campaign against them. This greatly impeded the technology infusion process.

Technology experts are essential to understanding and applying new technology correctly; but not all are well-suited to the advocate role. Advocates should be chosen carefully and the other technology experts kept in the background. Outside consultants should be used for initial training and coaching, and respected senior personnel and project leaders should be relied on to be coaches after they have been trained and have used the technology on a project.

- *Initial language training is best accomplished by outside vendors. Local training should focus on how to apply the language in the local environment.*

Of the two methods used for institutional Ada training in the FDD, the language courses taught by outside vendors (external to the local FDD/contractor organizations) were more successful. The FDD training experiences indicate that new technology training is best when taught by an instructor who is not known within the organization. That way the technology is not loaded with the extra baggage of personality conflicts or issues such as contractors teaching customers with whom they work on a daily basis. Obviously, local application of the technology should be taught by someone within the local organization. Here it is best to use a senior developer or manager who has learned the technology and applied it on a project, rather than a technology expert who may lack "real-world" experience using the technology.

Recommendations

- *The FDD should continue to use Ada whenever possible. This would include for those systems that reuse existing Ada code and any other projects (or portions of projects) that are expected to be long-lived and can be developed and deployed on an Ada-capable platform.*

Because many of the intended benefits of Ada have already accrued at the FDD and because the mainframe obstacle continues to hamper the complete adoption of Ada, it would be unrealistic for the FDD to mandate the use of Ada for all software development at the FDD.

However, it is also recommended that the FDD choose to use Ada in all cases where no clear disadvantage in doing so exists. This would indicate not only the continued use of Ada on satellite simulators but also the use of Ada on portions of any other projects that are expected to be long-lived and can be developed and deployed on an Ada-capable platform. As the FDD migrates away from mainframes and toward workstations, this will be an increasingly large segment of the software developed. Over the long term, Ada is a good candidate for future versions of the large reusable software libraries that are currently written in FORTRAN and maintained by a separate group of experts who continously augment the code's functionality to keep up with the needs of the client projects. Ada can be used to implement those subsystems, along with many other basic domain functions, as sets of separable and more maintainable abstractions, which would eliminate the high coupling found in the FORTRAN versions and lead to reduced maintenance costs.

- *The FDD should build reusable software in a language that supports object-oriented constructs and consider using specialized teams of experts to configure the reusable components for each mission. This would likely further improve the efficiency of the reuse process.*

The different reuse approaches used on the Ada and FORTRAN projects both had advantages and disadvantages. The best features of each should be combined to produce a more efficient reuse process. Reusable software components and architectures ideally would be implemented in a language that supports OO constructs, such as generics and strong typing, as does Ada. This will eliminate the size inflation experienced by using FORTRAN to emulate them and make the system more maintainable (less effort to enhance for future missions). However, the concept of a separate maintenance team for reusable software (as is currently used for the FORTRAN systems) should be retained. This will eliminate the need for project personnel to understand the complexities of generic architectures and parameterization (difficulties encountered by each Ada team in the FDD). It will also eliminate the configuration management risks associated with multiple mission-specific versions of the reusable software—risks that will increase as systems grow larger and live longer. Using this combined approach, the cost of the separate maintenance group would be expected to be much lower.

- *The FDD should investigate lower-cost alternative languages to support object-oriented development on workstations. However, trade-off analyses should consider the cost of software development environments, the efficiency and quality of software development, and the ease and cost of long-term maintenance for the languages under consideration.*

Over the next 5–10 years, the FDD will transition from the mainframes to open systems; future development will be done on workstations. The FDD's recent experience on both Ada and FORTRAN projects has demonstrated that object-oriented concepts lead to high levels of reuse. Because FORTRAN implementation of object-oriented (generalized) designs results in larger, more cumbersome systems and Ada development environments for workstations are somewhat expensive, neither Ada nor FORTRAN may be a practical language of choice for *all* future projects. Further SEL-conducted experiments are recommended to assess the suitability of one or more lower-cost alternative OO languages. Experiment results can be compared against the Ada and FORTRAN baselines documented in this report. Care should be taken, however, to consider the long-term implications of a language choice, not simply software development project results. For example, the cost savings of purchasing a C^{++} development environment instead of one to support Ada could be offset or absorbed by the extra cost to maintain C^{++} software.

Note to Readers Outside the FDD

One of the original objectives behind the DoD's development of the Ada language was the goal of providing a common language that would support the portability of programs, tools, and personnel across many projects. Another Ada goal was to provide, in Ada, a tool beneficial for large-system development and long-term maintenance. Because the FDD uses a single language and develops small to mid-sized systems with relatively short life spans, this organization was not able to assess Ada in the context for which it was designed. Hence, readers of this evaluation should bear in mind that this study reports only one experience with this technology. As the findings suggest, the language offers clear benefits and involves significant investment. The specific influential factors in any one organization (e.g., software domain, hardware environment, long-term goals) must be considered in any evaluation of Ada's applicability and effectiveness.

Appendix A. Project Data

Tables in this appendix present the project data used in the quantitative analysis in section 3 and the 1985 SEL baseline measures against which change was measured. The project data are from Ada and FORTRAN projects active in the FDD during the study period.

- Table A-1. Project Size Data
- Table A-2. Project Reuse Data
- Table A-3. Project Effort Data
- Table A-4. Characteristics and Schedule Data
- Table A-5. System Run-Time Performance Data
- Table A-6. Project Error Data
- Table A-7. 1985 SEL Baseline Measures

Table A-1. Project Size Data

Project Name	Type	Language	Total Lines	Comment Lines	Blank Lines	Nonblank/ Noncomment Lines	Total Statements	Declarations	Executables	Lines per Statement
GRODY	DS	Ada	128261	40462	27571	60228	20791	6846	13945	6.17
GOADA	DS	Ada	171102	41940	33450	95712	27776	10551	17225	6.16
GOESIM	TS	Ada	92095	26635	0	65460	17792	7098	10694	5.18
UARSTELS	TS	Ada	68148	17812	12021	38315	14674	7830	6844	4.64
EUVETELS	TS	Ada	66696	17482	11774	37440	14929	7900	7029	4.47
EUVEDSIM	DS	Ada	184017	46260	26685	111072	32544	14426	18118	5.65
SAMPEXTS	TS	Ada	61447	16957	10851	33639	13930	7271	6659	4.41
POWITS	TS	Ada	68107	18789	12544	36774	14909	6671	8238	4.57
TOMSTELS	TS	Ada	52295	14431	9235	28629	11855	6188	5667	4.41
FASTELS	TS	Ada	64723	17861	11430	35432	14168	7155	7014	4.57
FAST-GTC	AGSS subs	Ada	20426	4527	4527	11372	9507	4963	4544	2.15
GROSS	DS	FORTRAN	51704	22409	0	29295	27642	11927	15715	1.87
GROAGSS	AGSS	FORTRAN	236393	106908	0	129485	106033	54910	51123	2.23
GROSIM	TS	FORTRAN	38950	18082	0	20868	17787	7174	10613	2.19
GOFOR	DS	FORTRAN	37043	18926	0	18117	15635	5925	9710	2.37
GOESAGSS	AGSS	FORTRAN	128859	106908	0	21951	45846	19517	26329	2.81
-UARS partial	AGSS partial	FORTRAN	303126	170351	6063	126712	98673	41138	57535	3.07
-ACME	AGSS partial	FORTRANc	34902	17934	698	16270	15688	8981	6707	2.22
UARSAGSS	AGSS total	FORTRAN/c	338028	188285	6761	142982	114361	50119	64242	2.96
UARSDSIM	DS	FORTRAN	106446	54256	2129	50061	51323	20369	30954	2.07
EUVEAGSS	AGSS	FORTRAN	283911	133644	5678	144589	84097	35934	48163	3.38
-SAMPEX	AGSS partial	FORTRAN	154509	82839	3090	68580	51023	23748	27275	3.03
-SAMPEXTP	AGSS partial	FORTRANc	19632	9550	393	9689	7481	4594	2887	2.62
SAMPEX AGSS	AGSS total	FORTRAN/c	174141	92389	3483	78269	58504	28342	30162	2.98
-WINDDV	AGSS partial	FORTRAN	15244	7799	305	7140	5826	2615	3211	2.62
-WINDPOPS	AGSS partial	FORTRAN	11388	7578	228	3582	3444	1644	1800	3.31
-WINDPOLR	AGSS partial	FORTRANc	175415	80208	3508	91699	52064	23068	28996	3.37
ISTP	AGSS total	FORTRAN/c	202047	95585	4041	102421	61334	27327	34007	3.29
TOMSAGSS	AGSS	FORTRAN	255047	116620	5101	133327	54967	12808	42159	4.64
FASTAGSS partial	AGSS	FORTRAN	159080	111356	3182	44542	31070	7240	23831	5.12

Table A-2. Project Reuse Data

Project Name	Type	Language	Total Lines	New Lines	Extensively Modified Lines	Slightly Modified Lines	Verbatim LInes	Percent Verbatim Reuse	Percent Total Reuse
GRODY	DS	Ada	128261	123935	1143	3037	146	0%	2%
GOADA	DS	Ada	171102	109807	12496	41750	7049	4%	29%
GOESIM	TS	Ada	92095	59783	5784	15078	11450	12%	29%
UARSTELS	TS	Ada	68148	38327	6114	12163	11544	17%	35%
EUVETELS	TS	Ada	66696	2161	371	5573	58591	88%	96%
EUVEDSIM	DS	Ada	184017	20859	36248	87415	39495	21%	69%
SAMPEXTS	TS	Ada	61447	0	3301	6120	52026	85%	95%
POWITS	TS	Ada	68107	12974	7980	20878	26275	39%	69%
TOMSTELS	TS	Ada	52295	0	1768	11306	39221	75%	97%
FASTELS	TS	Ada	64723	2754	2552	17801	41616	64%	92%
FAST-GTC	AGSS subs	Ada	20426	16872	1378	1967	209	1%	11%
GROSS	DS	FORTRAN	51704	33196	3493	8574	6441	12%	29%
GROAGSS	AGSS	FORTRAN	236393	194169	9982	18133	14109	6%	14%
GROSIM	TS	FORTRAN	38950	31775	0	4294	2881	7%	18%
GOFOR	DS	FORTRAN	37043	22175	2867	6671	5330	14%	32%
GOESAGSS	AGSS	FORTRAN	128859	106834	6377	9779	5869	5%	12%
-UARS partial	AGSS partial	FORTRAN	303126	260382	9340	21536	11868	4%	11%
-ACME	AGSS partial	FORTRANc	34902	34902	0	0	0	0%	0%
UARSAGSS	AGSS total	FORTRAN/c	338028	295284	9340	21536	11868	4%	10%
UARSDSIM	DS	FORTRAN	106446	63861	17476	20710	4399	4%	24%
EUVEAGSS	AGSS	FORTRAN	283911	41552	13597	14844	213918	75%	81%
-SAMPEX	AGSS partial	FORTRAN	154509	10590	1631	1282	141006	91%	92%
-SAMPEXTP	AGSS partial	FORTRANc	19632	15899	1920	1777	36	0%	9%
SAMPEX AGSS	AGSS total	FORTRAN/c	174141	26489	3551	3059	141042	81%	83%
-WINDDV	AGSS partial	FORTRAN	15244	13471	581	196	996	7%	8%
-WINDPOPS	AGSS partial	FORTRAN	11388	7961	0	232	3195	28%	30%
-WINDPOLR	AGSS partial	FORTRANc	175415	142737	1581	25310	5787	3%	18%
ISTP	AGSS total	FORTRAN/c	202047	164169	2162	25738	9978	5%	18%
TOMSAGSS	AGSS	FORTRAN	255047	58217	2094	3467	191269	75%	76%
FASTAGSS partial	AGSS	FORTRAN	159080	27529	1985	9932	119634	75%	81%

Table A-3. Project Effort Data

Project Name	Mission	Type	Language	Effort[1,2]	Library[3] Task	Corrected Effort	Design Hours	Coding Hours	Testing Hours	Other Hours	All Activity Hours (no mgmt.)
GRODY	GRO	DS	Ada	23244	0	23244	4909	6467	2925	4832	19133
GOADA	GOES	DS	Ada	28056	0	28056	4967	7209	6131	7579	25886
GOESIM	GOES	TS	Ada	13658	0	13658	2503	2973	3081	4483	13040
UARSTELS	UARS	TS	Ada	11526	0	11526	2160	3067	3715	2226	11168
EUVETELS	EUVE	TS	Ada	4727	0	4727	644	711	1111	1771	4237
EUVEDSIM	EUVE	DS	Ada	20775	0	20775	3732	5348	3807	4918	17805
SAMPEXTS	SAMPEX	TS	Ada	2516	0	2516	341	338	546	814	2039
POWITS	Wind-Polar	TS	Ada	11695	0	11695	1072	2209	4760	3636	11677
TOMSTELS	TOMS	TS	Ada	3839	0	3839	905	852	219	773	2749
FASTELS	FAST	TS	Ada	6039	0	6039	1320	875	600	1324	4119
FAST-GTC	FAST	AGSS subs	Ada	4322	0	4322	1594	1223	469	106	3392
GROSS	GRO	DS	FORTRAN	15334	0	15334	3534	4253	2615	4762	15164
GROAGSS	GRO	AGSS	FORTRAN	54755	0	54755	10829	15642	11124	16283	53878
GROSIM	GRO	TS	FORTRAN	11463	0	11463	2408	3560	1681	3285	10934
GOFOR	GOES	DS	FORTRAN	12804	0	12804	1427	2260	4792	3754	12233
GOESAGSS	GOES	AGSS	FORTRAN	37806	0	37806	9256	11610	8976	6702	36544
-UARS partial	UARS	AGSS partial	FORTRAN	89514	0	89514	20561	24940	24710	15465	85676
-ACME	UARS	AGSS partial	FORTRANc	7965	0	7965	2195	1320	2370	1693	7578
UARSAGSS	UARS	AGSS total	FORTRAN/c	97479	0	97479	22756	26260	27080	17158	93254
UARSDSIM	UARS	DS	FORTRAN	17976	0	17976	3117	5831	4707	3542	17197
EUVEAGSS	EUVE	AGSS	FORTRAN	21658	0	21658	4419	5133	6437	4551	20540
-SAMPEX	SAMPEX	AGSS partial	FORTRAN	4598	3476	8074	654	290	1371	2185	4500
-SAMPEXTP	SAMPEX	AGSS partial	FORTRANc	6772	0	6772	1802	697	2620	1521	6640
SAMPEX AGSS	SAMPEX	AGSS total	FORTRAN/c	11370	3476	14846	2456	987	3991	3706	11140
-WINDDV	Wind-Polar	AGSS partial	FORTRAN	5952	0	5952	0	0	0	0	0
-WINDPOPS	Wind-Polar	AGSS partial	FORTRAN	2630	0	2630	0	0	0	0	0
-WINDPOLR	Wind-Polar	AGSS partial	FORTRANc	58010	0	58010	16661	6750	12526	9499	45436
ISTP	Wind-Polar	AGSS total	FORTRAN/c	66592	0	66592	16661	6750	12526	9499	45436
TOMSAGSS	TOMS-EP	AGSS	FORTRAN	13553	5103	18656	5259	3041	783	2806	11889
FASTAGSS partial	FAST	AGSS	FORTRAN	8502	6341	14843	2474	1477	694	1469	6113

1. The total effort hours for projects TOMSTELS, FASTELS, TOMSAGSS, and FASTAGSS were adjusted to compensate for a different testing process that was used on them. On the earlier projects, acceptance testing hours were not recorded; only developer hours needed to fix errors and finalize documentation were recorded. On these recent projects, the new independent testing process included both system and acceptance testing; thus, both were measured. Only one-half of the independent testing hours were included here as a good approximation of the system testing effort. This provides more comparable effort measurements.

2. The TOMSTELS and TOMSAGSS projects were stopped during the design phases and then restarted with substantially revised requirements. Their effort hours reflect only the effort spent after the project restart.

3. The MTASS (library task) hours per project were computed by summing the weekly effort hours reported during the life of each client project. The weekly effort was evenly allocated among the active client projects that MTASS was supporting in a given week. MSASS supported only one client project. Each MTASS/MSASS project total was further adjusted to 90% of the actual reported hours, so that only the effort spent enhancing the reusable components for the client projects would be included. This eliminated the hours spent fixing problems reported by ongoing mission users.

Table A-4. Characteristics and Schedule Data

Project Name	Mission	Type	Language	Begin Date	End Date	Duration (months)
GRODY	GRO	DS	Ada	6/29/85	10/1/88	39
GOADA	GOES	DS	Ada	6/6/87	4/14/90	34
GOESIM	GOES	TS	Ada	9/5/87	7/29/89	23
UARSTELS	UARS	TS	Ada	2/13/88	12/2/89	22
EUVETELS	EUVE	TS	Ada	10/1/88	5/5/90	19
EUVEDSIM	EUVE	DS	Ada	10/1/88	1/26/91	28
SAMPEXTS	SAMPEX	TS	Ada	3/31/90	3/2/91	11
POWITS	Wind-Polar	TS	Ada	3/24/90	5/9/92	26
TOMSTELS	TOMS	TS	Ada	12/12/92	9/30/93	10
FASTELS	FAST	TS	Ada	8/1/92	10/23/93	15
FAST-GTC	FAST	AGSS subs	Ada	8/4/92	4/29/94	21
GROSS	GRO	DS	FORTRAN	12/29/84	10/10/87	33
GROAGSS	GRO	AGSS	FORTRAN	8/3/85	3/11/89	43
GROSIM	GRO	TS	FORTRAN	8/31/85	8/1/87	23
GOFOR	GOES	DS	FORTRAN	6/6/87	9/16/89	27
GOESAGSS	GOES	AGSS	FORTRAN	8/29/87	11/11/89	26
-UARS partial	UARS	AGSS partial	FORTRAN	11/21/87	9/15/90	34
-ACME	UARS	AGSS partial	FORTRANc	1/30/88	9/15/90	32
UARSAGSS	UARS	AGSS total	FORTRAN/c	11/21/87	9/15/90	34
UARSDSIM	UARS	DS	FORTRAN	1/2/88	6/16/90	29
EUVEAGSS	EUVE	AGSS	FORTRAN	10/1/88	9/15/90	23
-SAMPEX	SAMPEX	AGSS partial	FORTRAN	3/31/90	11/16/91	20
-SAMPEXTP	SAMPEX	AGSS partial	FORTRANc	3/31/90	11/30/91	20
SAMPEX AGSS	SAMPEX	AGSS total	FORTRAN/c	3/31/90	11/30/91	20
-WINDDV	Wind-Polar	AGSS partial	FORTRAN	9/29/90	1/2/93	27
-WINDPOPS	Wind-Polar	AGSS partial	FORTRAN	6/23/90	5/9/92	23
-WINDPOLR	Wind-Polar	AGSS partial	FORTRANc	2/10/90	8/15/92	30
ISTP	Wind-Polar	AGSS total	FORTRAN/c	2/10/90	1/2/93	35
TOMSAGSS	TOMS-EP	AGSS	FORTRAN	2/5/93	4/15/94	14
FASTAGSS partial	FAST	AGSS	FORTRAN	8/1/92	4/29/94	21

1. The TOMSTELS and TOMAGSS projects were stopped during the design phases and then restarted with substantially revised requirements. Their dates have been adjusted to remove the time gap.

Table A-5. System Run-Time Performance Data

Project Name	Mission	Type	Language	Simulated Time to Clock Time	Clock Hours per Simulated Hour
GOADA	GOES	DS	Ada	0.50	2.00
GOESIM	GOES	TS	Ada	3.00	0.33
UARSTELS	UARS	TS	Ada	2.55	0.39
EUVETELS	EUVE	TS	Ada	2.60	0.38
SAMPEXTS	SAMPEX	TS	Ada	8.00	0.13
POWITS	Wind-Polar	TS	Ada	0.50	2.00
TOMSTELS	TOMS	TS	Ada	10.00	0.10
FASTELS	FAST	TS	Ada	6.00	0.17
GROSIM	GRO	TS	FORTRAN	0.65	1.54
COBESIM	COBE	TS	FORTRAN	2.00	0.50
GOFOR	GOES	TS	FORTRAN	8.00	0.13

Table A-6. Project Error Data

Project Name	Type	Language	Total Lines (SLOC)	Developed Lines (DLOC)	Statements	Development Errors	Errors per KSLOC	Errors per KDLOC	Errors per Statement
GRODY	DS	Ada	128,261	125,715	20,791	229	1.8	1.8	11.0
GOADA	DS	Ada	171,102	132,063	27,776	415	2.4	3.1	14.9
GOESIM	TS	Ada	92,095	70,873	17,792	127	1.4	1.8	7.1
UARSTELS	TS	Ada	68,148	49,182	14,674	153	2.2	3.1	10.4
EUVETELS	TS	Ada	66,696	15,365	14,929	9	0.1	0.6	0.6
EUVEDSIM	DS	Ada	184,017	82,489	32,544	128	0.7	1.6	3.9
SAMPEXTS	TS	Ada	61,447	14,930	13,930	10	0.2	0.7	0.7
POWITS	TS	Ada	68,107	30,385	14,909	84	1.2	2.8	5.6
TOMSTELS	TS	Ada	52,295	11,873	11,855	6	0.1	0.5	0.5
FASTELS	TS	Ada	64,723	17,189	14,168	30	0.5	1.7	2.1
FAST-GTC	AGSS subs	Ada	20,426	18,685	9,507	15	0.7	0.8	1.6
GROSS	DS	FORTRAN	51,704	39,692	27,642	104	2.0	2.6	3.8
GROAGSS	AGSS	FORTRAN	236,393	210,599	106,033	931	3.9	4.4	8.8
GROSIM	TS	FORTRAN	38,950	33,210	17,787	296	7.6	8.9	16.6
GOFOR	DS	FORTRAN	37,043	27,442	15,635	147	4.0	5.4	9.4
GOESAGSS	AGSS	FORTRAN	128,859	116,341	45,846	603	4.7	5.2	13.2
-UARS partial	AGSS partial	FORTRAN	303,126	276,403	98,673	778	2.6	2.8	7.9
-ACME	AGSS partial	FORTRANc	34,902	34,902	15,688	175	5.0	5.0	11.2
UARSAGSS	AGSS total	FORTRAN/c	338,028	311,305	114,361	953	2.8	3.1	8.3
UARSDSIM	DS	FORTRAN	106,446	86,359	51,323	613	5.8	7.1	11.9
EUVEAGSS	AGSS	FORTRAN	283,911	100,901	84,097	123	0.4	1.2	1.5
-SAMPEX	AGSS partial	FORTRAN	154,509	40,679	51,023	31	0.2	0.8	0.6
-SAMPEXTP	AGSS partial	FORTRANc	19,632	18,182	7,481	73	3.7	4.0	9.8
SAMPEX AGSS	AGSS total	FORTRAN/c	174,141	58,860	58,504	104	0.6	1.8	1.8
-WINDDV	AGSS partial	FORTRAN	15,244	14,290	5,826	65	4.3	4.5	11.2
-WINDPOPS	AGSS partial	FORTRAN	11,388	8,646	3,444	38	3.3	4.4	11.0
-WINDPOLR	AGSS partial	FORTRANc	175,415	150,537	52,064	959	5.5	6.4	18.4
ISTP	AGSS total	FORTRAN/c	202,047	173,474	61,334	1062	5.3	6.1	17.3
TOMSAGSS	AGSS	FORTRAN	255,047	99,258	54,967	62	0.2	0.6	1.1
FASTAGSS partial	AGSS	FORTRAN	159,080	55,427	31,070	57	0.4	1.0	1.8

Table A-7. 1985 SEL Baseline Measures*

Productivity	26 SLOC/day 11.8 statements/day	
Code Reuse	20%	
Error Rate	6.5 errors/KSLOC 14.3 errors/thousand statements	
Maintenance Cost	8–15% of the development cost per year	
Effort Distribution	Design Code Test Other	23% 21% 30% 26%
Classes of Errors	Data Interface Logic/Control Initialization Computational	27% 22% 20% 16% 15%
Project Duration	Simulators ~ 20 months AGSS ~ 30 months	

*Based on FDD projects active between 1978 and 1985

Appendix B. Detailed Reuse Analysis

During the last year of the independent assessment, the team delved more deeply into the reuse issue. They sought to understand the different reuse approaches that have been used on the FORTRAN and Ada projects and to determine their effect on the resulting products. Ultimately, they hoped to determine which improvements resulted from the different approaches for managing and modifying the reusable code and which ones were due to differences and limitations in the languages. This section presents the detailed results of this analysis, some of which have been highlighted in section 3 of this report. The complete information is included here, because this reuse analysis has not been documented elsewhere. The following subsections provide insight into the different approaches to reuse, reuse library maintenance costs, the effect of different kinds of reuse on productivity, and the validity of the currently used cost model for Ada projects.

B.1 Ada vs. FORTRAN Reuse Methods

The reuse approach used to develop the FORTRAN AGSS systems differs from that used to develop the Ada telemetry simulator projects. In the FORTRAN projects, the application programmer links to reusable subsystems from one of two large subsystem families (known as MTASS, for multimission three-axis-stabilized spacecraft, and MSASS, for multimission spin-axis-stabilized spacecraft) and adds new modules which conform to the data set specifications preordained by those subsystem interfaces. Maintenance programmers for the libraries of reusable subsystems are tasked to modify and update the master copy of each subsystem so that a single version of each can meet the requirements of new missions while at the same time satisfying all previous client missions. This backwards compatibility is important because AGSS systems must stay operational for the duration of a satellite mission, which can be many years. Thus, the libraries are considered to be institutional software and maintenance is funded independently from the development of each of the mission-specific AGSS projects.

The existence of single, centralized copies of the library subsystems for all users means that the individual application programmers for each project do not have to be concerned with the internals of the subsystems; these components are not copied into each project library and are therefore not treated as mission-specific code. This lowers the development burden by reducing the amount of code that must be handled by a project. For example, the SAMPEX AGSS client application required only 27K new lines of code to be written but reported a delivered size of 176K lines because of the subsystems it reused from MTASS.

The Ada projects, on the other hand, are constructed from generic components copied into each new project library from the most similar prior project. Most of these components remain unmodified, although modification by a project team is permissible because it poses no risk to central, shared copies. The project team therefore has the burden of directly handling and studying the generics to determine their suitability and possible need for modification. Any maintenance is the job of the project team since there is no separate dedicated team responsible for upgrading a central copy of the components to meet new requirements. Further, there is no comprehensive documentation of the generics or of the general telemetry simulator architecture that might compensate for the lack of an expert library maintenance team and that might allow an Ada programming team to reuse components in the same "black-box" fashion that the FORTRAN developers are able to do.

Because Ada allows more parameterization and generalization than FORTRAN, the reusable Ada generics take advantage of this additional flexibility by allowing mission-tailored, instead of hard-coded, data sets to be defined and passed among the subsystems. The different styles of reuse and the higher generality of the Ada components explain why there was a milder drop in reuse when the change of domains occurred in the Ada product line as compared with the FORTRAN projects. The Ada project team for the first spin-axis-stabilized mission was still able to reuse a sizable portion of the lower-level three-axis generics, whereas almost none of the FORTRAN subsystems that handle three-axis missions were used for the first spin-axis satellite. The FORTRAN developers again achieved high levels of reuse in their projects by developing a separate complete subsystem library for

spin-axis spacecraft (MSASS) analogous to the library of three-axis subsystems (MTASS). Instead of having controlled libraries of subsystems, the Ada systems themselves became the basis for future simulators in either domain.

An important distinction between the reuse styles adopted for the two languages is that the two FORTRAN libraries must be continually augmented to handle new missions in their respective domains. It is the practice of the FORTRAN maintainers to augment the subsystems as necessary by adding code for any new requirements rather than by generalizing or modifying the existing code. This approach is more straightforward given the limitations of FORTRAN and it also avoids the risk of introducing errors for existing clients. However, this also causes the FORTRAN libraries to grow over time. Conversely, the Ada generics form a set of smaller components that requires little or no further modification to handle missions in either domain. The Ada developers directly handle the generics needed for each project and further generalize them only when necessary (such as by deleting unnecessary dependencies between components). New requirements (which typically involve the simulation of new spacecraft sensors and devices) are handled by mission-specific code rather than by changes to the reusable components.

Because the separate effort to upgrade the FORTRAN subsystems is not reported by the individual projects, the verbatim reuse percentages reported in the project data make it appear that the two languages are equally able to express generalized functionality. However, further investigation revealed that, when the efforts of the separate FORTRAN maintenance programmers are taken into consideration, the actual amount of modification to the Ada generics from mission to mission is far less than in the FORTRAN subsystems.

Maintenance and configuration control disadvantages can result from having separate copies of the reusable components in each client project's library. However, this has not been an issue with the Ada simulators, which are smaller and have shorter operational phases than the larger AGSS projects. Nevertheless, this approach introduces the cost of directly handling this software and means that the developers, with neither a library support team nor comprehensive documentation (as yet), must study the internals of the reusable components to understand their proper use and to determine if any enhancements are needed. The additional cost for this aspect of Ada reuse is calculated through the comparison of "white-box" and "black-box" reuse presented in section B.3.

B.2 Adjusting for the FORTRAN Library Maintenance Costs

Because the additional effort expended on the part of the MTASS and MSASS library maintenance tasks benefits each of the client FORTRAN AGSS projects, it is necessary to consider these hours when reporting the overall costs and productivities of the recent AGSS projects. However, there is no entirely accurate way to apportion the MTASS and MSASS hours across the client projects; effort data are collected at an inadequate level of detail to capture that information. Nevertheless, a rough idea can be obtained by using that part of the MTASS/MSASS effort that is spent doing enhancements. The total MTASS/MSASS effort for each client project can be calculated by totaling the reported weekly effort data (evenly allocated among all active projects) for the appropriate maintenance group during the time the client project was in active development. SEL data show that 90% of the MTASS/MSASS effort is spent doing enhancements; therefore each project's MTASS/MSASS effort contribution can be reduced to 90% of the total. Thus, the additional effort expended on behalf of each project using this allocation method ranges from about 3.5K hours to more than 6K hours, with the later projects showing greater maintenance costs.

It is important, when assessing total cost, to include the library maintenance effort in the FORTRAN project totals. To fail to do so would seriously underrepresent the actual cost of the FORTRAN AGSS development. On the other hand, when using the data to model the cost of new, modified, or reuse-based development from the project point of view, only the reported mission-specific effort should be used.

B.3 Computing the Productivity of Reuse

Conventionally in this environment, reuse is classified as either verbatim reuse or reuse with modification. Using the technique developed by Bailey,[17] individual productivities of the different categories or modes of code development/reuse in the FDD were estimated by deriving a set of simultaneous equations and then solving for the unknown productivities. The effort for each project was expressed as the sum of the efforts to develop the various

amounts of code in each category (new, modified, verbatim). With sufficient data, it is possible to solve for the set of productivities for the modes which most closely predict the actual effort required for each project.

A similar analysis was conducted for the Ada projects in the current study, and the results were comparable to Bailey's earlier work. Also, a similar analysis was performed to solve for the corresponding productivities on the FORTRAN projects. The best overall solutions for the productivities for new, reused with modification, and verbatim reuse for both Ada and FORTRAN code are shown in Table B-1. The FORTRAN solutions were not as stable as the Ada solutions, and they had to be constrained to prevent anomalous results.

Table B-1. FORTRAN vs. Ada Productivities (Statements per Hour) for Code Development Modes

Category of Code Reuse	FORTRAN	Ada
New Code	1.2	1.1
Reuse with Modification	2.4	1.2
Verbatim Reuse	5.5	5.0

In the table, the productivities for both languages are nearly identical except for the "reuse with modification" category, where the FORTRAN productivity is double that of Ada. This could indicate that FORTRAN units are easier to modify than Ada units. However, this analysis concludes that the difference actually reflects the learning curve required for reusing generic Ada code. When a project team needs to modify a part of the reusable software, additional effort is required first to understand the code and its applicability, and then to generalize it further to ensure future reusability.

As mentioned in section B.2, the verbatim reuse category in the FORTRAN projects denotes a different reuse process than the verbatim reuse being performed in Ada. Ever since the availability of MTASS and MSASS, the majority of the code reported by a FORTRAN project as reused verbatim has been from the reusable library subsystems which are managed externally from the application developments. Instead of deducting the amount of software contributed by the external libraries and reanalyzing the productivities for each development mode on the remaining project-specific software, an additional mode of code development was defined: "black-box" verbatim reuse. This is the reuse of software from externally maintained libraries, where the application programmer is not required to learn or to pay attention to the internals of the reused code. This is in contrast to the alternate method of verbatim reuse where the application programmer is responsible for deciding whether a particular component is appropriate and reusable by studying and understanding its implementation. The term "white-box" has been adopted to describe this style of verbatim reuse. Separating the verbatim reuse in this way allows a better approximation of the overhead involved in learning, understanding, integrating, and testing software that can be reused without change. It also provides a more equivalent basis for comparing the cost of verbatim reuse across the languages.

Separation of the verbatim reuse category into black-box and white-box reuse for the later FORTRAN AGSS projects where MTASS and MSASS were used yielded a more stable and well-behaved set of productivity estimates for the development modes. As one might expect, the productivity for the new black-box verbatim reuse category was very high. Depending on the group of projects included in the solution, some of the analyses showed it to be essentially "infinite" (meaning that black-box statements can be "developed" for free, so the size of the reused components has little or no effect on the reusing project's cost). This means that productiviy values for the other categories would be unaffected even if the black-box verbatim statements were eliminated from the project totals.

Reuse-library-supplied statements were included because the current reporting style is to include them in project totals. However, in the future it might make more sense to exclude them from project development estimates and reported sizes, analogous to the way the size of a math library is ignored. It would still be important to budget for the library maintenance task, however, and to understand that library maintenance remains an additional cost of delivering FORTRAN AGSS projects. Eliminating the reporting of the FORTRAN library software which

masquerades as zero-cost verbatim reuse would also bring the Ada and FORTRAN reuse factors more in line with one another.

The productivities for the FORTRAN development modes with the addition of black-box verbatim reuse are shown in Table B-2. There is no development mode corresponding to black-box verbatim reuse on the Ada projects.

Table B-2. Code Development Productivities Including Black-Box Reuse

Category of Code Reuse	FORTRAN	Ada
New Code	1.2	1.1
Reuse with Modification	2.4	1.2
White-Box Verbatim Reuse	4.0	5.0
Black-Box Verbatim Reuse	21.0	N/A

B.4 Comparing Reuse Factors with Existing Models

The current model used in the FDD for estimating the cost of reuse was developed based on empirical data available in 1993.[11] It specifies that development by reuse in FORTRAN costs about 20% of the cost of new code development, but that reuse in Ada costs about 30% of the cost of new code. These figures are the "reuse factors" for each language that can be multiplied by the new code development costs to estimate the cost of delivering reused software. This model suggests that it costs 50% more to reuse Ada over FORTRAN from the reusing project's point of view.

The findings in this report suggest that the apparent advantage that FORTRAN reuse has over Ada reuse is created by the highly productive black-box verbatim reuse used on FORTRAN projects, which is not available to the Ada projects. The cost of the separate task which offloads the actual expense of the black-box code (i.e., the effort to understand and modify the FORTRAN utility subsystems) is not included in these reuse cost estimates because it is funded separately and available to all FORTRAN AGSS projects. However, because the separately funded cost of maintaining the reusable libraries raises the true cost of the FORTRAN projects in a way that is not reflected by these models, the reuse cost factors are not directly comparable.

A better way to look at the relative costs of reuse in the two languages is to consider the ratios of productivities between new and reused code in each language, as was done in section B.3. These ratios appear to be nearly identical (except for reuse with modification, where, in the FORTRAN case, a separate team performs the modifications, and the productivity rates diverge accordingly), which suggests that similar reuse processes result in similar productivity levels, regardless of language. In fact, it even appears that the per-line productivities are comparable between the languages, which should further simplify future cost models.

Appendix C. Data Collection Instruments

As part of the independent assessment, the team attempted to capture and understand the perspectives of the FDD software engineering staff. They conducted two surveys to gather this information. The first addressed only those who had been directly exposed to Ada through work experience or training. The second addressed the entire workforce. This appendix includes copies of these two data collection instruments.

- Figure C-1. Ada User's Survey
- Figure C-2. Ada at the FDD Questionnaire

1. How many years have you been a part of the FDD?

2. What was your principal training prior to joining the FDD? (astrodynamics, simulation, computer science, mathematics, engineering, etc...)

3. How many years of FORTRAN work experience did you have before joining the FDD? How many school (college) years?

4. How many years of Ada work experience did you have before joining the FDD? How many school years?

5. How many years of C language work experience did you have before joining the FDD? How many school years?

6. In what other languages can you (or could you at one time) program?

7. If you were leading the team to develop the next simulator, what language would you use? Why? Under what circumstances would you use either of the other two languages?

8. If you were leading the team to develop the next AGSS, what language would you use? Why? Under what circumstances would you use either of the other two languages?

9. What other opinions do you have relative to the use of Ada, FORTRAN, or C in FDD software developments? (Any comments on the impact of team size, schedule constraints, reuse constraints, cost constraints, maintainability constraints, portability contraints, performance constraints, etc.?)

Figure C-1. Ada User's Survey

Ada at the FDD Questionnaire

1. Name (optional, but please see note at bottom)_____

2. For how many years have you been a part of the FDD? _____

3. What is your primary job? *Circle one:* Developer Maintainer Tester Manager Other:_____

4. What was your principal professional training or education major prior to joining the FDD?
   Circle or write-in:

astrodynamics	astronomy	aerospace	business	chemistry
computer science	control theory	engineering	geophysics	numerical science
mathematics	music	physics	simulation	operations research
other: _____			_____	

5. How many years of FORTRAN work experience, including school, have you had?_____

6. Have you ever been exposed to the Ada programming language? *Circle one:* Yes* No**

   *If "yes," approximate year of first exposure: 19_____

   *If "yes," circle kinds of Ada experience:

professional training/classes/seminars	school	self-study
on-the-job development or maintenance	other_____	other _____

   **If "no," circle the sources of information which have contributed the most to your impressions of Ada:

news items	professional literature	others in the FDD	others outside of the FDD
job experiences	conferences	other_____	other_____

7. In what languages other than Ada or FORTRAN can you (or could you at one time) program?
   Circle or write-in:

Algol	Assembly	APL	Basic	C	C++	COBOL
Euclid	Lisp	Modula	Pascal	PL-1	Prolog	Smalltalk
Snobol	_____		_____		_____	

Note: Opinions or comments provided will only be associated with groups, such as developers, testers, etc., and will not be associated with individuals. Your name, in answer to question 1, will be known only by the outside consultant and will be used only in case it is necessary to contact you for clarification. These sheets will be kept off site until the final report is completed, after which they will be destroyed. Thank you for your time and cooperation.

Figure C-2. Ada at the FDD Questionnaire (1 of 2)

8. At the end of this study, a final report will be issued which will include discussion of the future of Ada at the FDD. *Circle your answers to the following and provide comments as indicated:*

a.	Do you think the use of Ada is appropriate at the FDD?	Yes	No	Don't know	Don't care
b.	Do you think the use of Ada should be restricted at the FDD? *If yes, describe appropriate and inappropriate uses of Ada:	Yes*	No	Don't know	Don't care
c.	Would you like to see the use of Ada increase at the FDD? *If yes, describe additional areas where Ada can be used:	Yes*	No	Don't know	Don't care
d.	Would you like to see the use of Ada decrease at the FDD? If yes, describe areas where Ada should be eliminated:	Yes*	No	Don't know	Don't care

9. Many FDD personnel have strong opinions about Ada and its use here. We are particularly interested in any comments and judgements you may have about the use of Ada at the FDD which you have not had the opportunity to express in your previous responses. Please use the space below to summarize, as frankly as possible, any additional opinions you have about Ada and its use at the FDD.

Figure C-2. Ada at the FDD Questionnaire (2 of 2)

Acronyms

AGSS	attitude ground support system
CDR	critical design review
COMPASS	Combined Mission Planning and Attitude Support System
DLOC	developed lines of code
DoD	Department of Defense
EMS	electronic message system
EUVE	Extreme Ultraviolet Explorer
EUVEAGSS	EUVE Attitude Ground Support System
EUVEDSIM	EUVE Dynamics Simulator
EUVETELS	EUVE Telemetry Simulator
FAST	Fast Auroral Snapshot Telescope
FAST GTC	FAST General Torquer Command Utility
FASTELS	FAST Telemetry Simulator
FDAS	Flight Dynamics Analysis System
FDD	Flight Dynamics Division
FDDS	Flight Dynamics Distributed System
GENSIM	Generalized Simulator
GOADA	GOES Dynamics Simulator in Ada
GOES	Geostationary Operational Environmental Satellite
GOESIM	GOES Telemetry Simulator
GOOD	General Object-Oriented Design
GRO	Gamma Ray Observatory
GRODY	GRO Dynamics Simulator in Ada
GROSS	GRO Dynamics Simulator in FORTRAN
GSFC	Goddard Space Flight Center
GSS	Generalized Applications Support Software
ISTP	International Solar-Terrestrial Physics
KSLOC	thousand source lines of code
MSASS	multimission spin-axis-stabilized spacecraft

MTASS	multimission three-axis-stabilized spacecraft
NASA	National Aeronautics and Space Administration
OO	object-oriented
OOD	object-oriented design
PDR	preliminary design review
POLAR	Polar Plasma Laboratory
POWITS	WIND/POLAR Telemetry Simulator
SAMPEX	Solar, Anomalous, and Magnetospheric Particle Explorer
SAMPEXTS	SAMPEX Telemetry Simulator
SEL	Software Engineering Laboratory
SLOC	source lines of code
TDRSS	Tracking and Data Relay Satellite System
TOMS	Total Ozone Mapping Spectrometer
TOMSTELS	TOMS Telemetry Simulator
TONS	TDRSS Onboard Navigation System
UARS	Upper Atmosphere Research Satellite
UARSTELS	UARS Telemetry Simulator
UIX	User Interface Executive
WIND	Interplanetary Physics Laboratory

References

1. NASA/GSFC Software Engineering Laboratory, SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. Basili, et al., December 1994

2. ___, SEL-93-003, "Impact of Ada in the Flight Dynamics Division: Excitement and Frustration," J.Bailey, S.Waligora, M. Stark, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, pp. 422–438, December 1993

3. ___, SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, L.Landis, R. Kester, November1993

4. ___, SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. McGarry, etal., June 1992

5. ___, SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

6. Computer Sciences Corporation, CSC/TM-91/6065 (552-FDD-91/034) *SEL Ada Reuse Study Report*, R. Kester, May 1991

7. NASA/GSFC Software Engineering Laboratory, SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory*, L. Jun and S. Valett, June 1990

8. ___, SEL-91-003, *Ada Performance Study Report*, E. Booth and M. Stark, July 1991

9. Goddard Space Flight Center, Flight Dynamics Division, 552-FDD-91/068R0UD0, *Ada Efficiency Guide*, E.Booth, August 1992

10. ___, 552-FDD-92/033 R0UD0 *Ada Size Study Report*, S. Condon, M. Regardie, September 1992

11. NASA/GSFC Software Engineering Laboratory, SEL-93-002, *Cost and Schedule Estimation Study Report*, S.Condon, M. Regardie, M. Stark, et al., November 1993

12. ___, SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

13. ___, SEL-91-006, "Experiments in Software Engineering Technology," *Proceedings of the Sixteenth Annual Software Engineering Workshop*, F. McGarry and S. Waligora, December 1991

14. ___, SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J.Bristow, D. Smith, November1994

15. Institute for Defense Analysis, IDA Paper P-2899, "*Comparing Ada and FORTRAN Lines of Code: Some Experimental Results*," T. Frazier, J. Bailey, M. Young, November1993

16. ANSI/MIL-STD 1815A, *Reference Manual for the Ada Programming Language*, February 1983

17. Bailey, John W., *A Component Factory for Software Source Code Re-engineering and Reuse*, University of Maryland, UMI 9234514, May 1992

18. Goddard Space Flight Center, Flight Dynamics Division, *Ada Compilers on the IBM Mainframe (NAS8040) Evaluation Report*, L. Jun, January 1989

19. ___, *IBM Ada/370 (Release 2.0) Compiler Evaluation Report*, L. Jun, September 1992, and *Intermetrics MVS/Ada Version 8.0 Compiler Evaluation Report*, L. Jun, October1992

Standard Bibliography of SEL Literature

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W.A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F.E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering,* V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data,* D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems,* G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop,* December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL),* A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection,* V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, etal., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P.Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry etal., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, etal., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D.Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 3)*, L.Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M.Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J.Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-002, *Software Measurement Guidebook*, M. Bassman, F. McGarry, R. Pajerski, July 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-95-001, *Impact of Ada and Object-Oriented Design in the Flight Dynamics Division at Goddard Space Flight Center*, S. Waligora, J. Bailey, M. Stark, March 1995

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[12]Basili, V., and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

[12]Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squire, "A Change Analysis Process to Characterize Software Maintenance Projects", *Proceedings of the International Conference on Software Maintenance*, September 1994

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[12]Briand, L., S. Morasca, and V. R. Basili, *Defining and Validationg High-Level Design Metrics*, University of Maryland, Technical Report TR-3301, June 1994

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May1993

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/ April1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[12]Seidewitz, E., "Genericity versus Inheritance Reconsidered: Self-Reference Using Generics," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS_92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11]This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12]This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Contractor Report

4. TITLE AND SUBTITLE

Impact of Ada and Object-Oriented Design in the Flight Dynamics Division at Goddad Space Flight Center

5. FUNDING NUMBERS

552

IN-61-TM
53150
P. 92

6. AUTHOR(S)

Software Engineering Laboratory

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland

8. PERFORMING ORGANIZATION REPORT NUMBER

SEL-95-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

NASA Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CR-189412

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category: 61
Report is available from the NASA Center for AeroSpace Information,
800 Elkridge Landing Road, Linthicum Heights, MD 21090; (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Laboratory (SEL) is an organization sponsored by NASA/GSFC and created to investigatethe effectiveness of software engineering technologies when applied to the development of applications software. The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

14. SUBJECT TERMS

Ada, Object-Oriented Design (OOD), Software Engineering

15. NUMBER OF PAGES

86

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited