STGT Program:
Ada Coding and
Architecture
Lessons Learned

Paul Usavage and Don Nagurney
GE Management and Data Systems Engineering,
P.O. Box 8048, Phila. PA, 19101
Phone: (215) 354–3165
Fax: (215) 354–3177

STGT (Second TDRSS Ground Terminal) is currently halfway through the System Integration Test phase (Level 4 Testing). To date, many software architecture and Ada language issues have been encountered and solved. This paper, which is a transcript of the presentation at the December 3rd meeting, attempts to define these lessons plus others learned regarding software project management and risk management issues, training, performance, reuse, and reliability. Observations are included regarding the use of particular Ada coding constructs, software architecture trade-offs during the prototyping, development and testing stages of the project and dangers inherent in parallel or concurrent systems, Software, Hardware and Operations Engineering.
Introduction

STGT is the first major Ada development program for M&DSO, which has developed other large ground stations in FORTRAN and C. In addition to the use of Ada, GE Management and Data Systems Operations faced other software development risks in the implementation of STGT. Some of these risk items are itemized below:

- A heavily distributed system (> 30 processing nodes and > 100 workstations in previous ground stations)
- High real-time system content (vs. 40% real-time, 60% batch processing)
- First on a DEC/VAX platform (vs. IBM mainframes and Sun/Unix workstations)
- High-availability/high-reliability architecture (99.99% availability required)
- High hardware content (> 350 racks of ground communication equipment)
- Heavily automated, X-Windows, workstation-based user interface
- First artificial intelligence (AI) based hardware fault detection/fault isolation
- Short development lead time (3 years from start to delivery)

Risk items like the above don’t usually translate into the impossible, they just have a way of eating into cost and schedule margins. Several steps were taken to mitigate the risks involved. An Ada Core Team was formed prior to program startup to develop language expertise. An Ada training program was developed and its completion required for all software engineers employed on the program. Despite these efforts, many lessons were learned on the job through prototyping, development and testing. This paper is intended to be a chronicle of these risk issues and (hopefully) their resolutions.

Project Composition

The Second TDRSS (Tracking and Data Relay Satellite System) Ground Terminal (STGT) is a new ground station and an upgrade to an existing ground station in White Sands, New Mexico. These ground stations will provide command and data communications from user control facilities through the TDRS, and on to the various user satellites and the Space Shuttle.

The breakdown of thousands of source lines of code developed for each Computer Software Configuration Item (CSCI) for the project is shown in Table 1.
Descriptions of the CSCIs are as follows:

- **TTC:** Tracking, Telemetry and Command
  CSCI, responsible for controlling the Tracking and Data Relay Satellites (TDRS) used by NASA to relay user satellite and space shuttle telemetry and command data. Responsible for commanding the satellite, monitoring its health, and controlling the ground antenna in order to point at the satellite.

- **DIS:** Data Interface Subsystem, responsible for interfacing with the NASA communication network, accepting scheduling orders from NASA, and switching the inputs and outputs from the ground station to data links between STGT and the other NASA locations.

- **USS:** User Services Subsystem, responsible for controlling most of the ground communications equipment (GCE) and supporting communications to the TDRS and to various user satellites.

- **EXC:** Executive, responsible for scheduling of a single Space to Ground Link Terminal (SGLT) controlling a single TDRS satellite. There will be six SGLTs overall in the two ground station installations.

- **WKS:** Workstation, responsible for operator interface, including intelligent graphically-oriented displays, operation alert messages and operator commanding capabilities.

- **COM:** Common Run-time Environment, provides common capabilities across all computers including communications within and between computers, data logging, startup/shutdown/failover control, and device driver interfaces.
Software Architectural Issues

Architectural Reuse

STGT attempted a high level of reuse and incorporated reuse into its architecture and in many ways succeeded. Attempts were made in object-oriented design, some of which succeeded in providing reliable, understandable, reusable products, and some of which only caused major headaches. Those that were problematic were usually related to lack of understanding of the scope and breadth of the situations in which the code would be reused, the computers on which the code would run, and the environments on these computers. For example, code reused on a workstation found itself in a rather different environment than on the large VAXes, due to lack of availability of large local databases.

Reuse was attempted on both large and small scales. Small-scale reuse was of course more easily planned than large-scale reuse. Large-scale reuse was more likely to result in complicated error conditions, where different subsystems (and their engineers and programmers) wanted to operate in different ways but were constrained by identical implementations due to code reuse.

Ada Reuse

In the early days we had "reuse evangelists" who proposed massive, complex, self-initializing generics for everyone. Almost every case that was ever implemented was later disabled, deleted, gutted or otherwise rewritten. Generics proved very difficult to debug using a source-level interactive debugger, relatively slow to execute in real-time, and very hard to write. Elaboration time-initialization code was also difficult to debug and prone to exception handling difficulties. Simple generics, on the other hand, were often very effective and easy to reuse. Complicated generics (including generics within generics within generics) were seldom worth the cost unless the designer and sole user were one-and-the-same, and the designer was well above-average in terms of proficiency and experienced at writing generics. That's not to say that we didn't have proficient programmers. With 100 or more programmers, just don't expect everyone to be a generics expert and design generics well.

Our best use of complex Ada generics involved data logging and retrieval software. This software utilized a high number of generics starting with primitive types (strings, integers, reals) and built up by instantiation
into complex, compound record structures in various sizes and formats. This worked very well, provided a single designer/programmer was responsible for both the generic capability and its uses.

Other good choices for Ada generics were design elements which clearly had a high degree of parallelism, such as our communications package which treated all messages the same, regardless of individual message formats. These even utilized declare blocks, which instantiated the generic on-the-fly for differing sizes or other record discriminants based on run-time values. These met with good success and surprisingly good performance on VAX Ada. Poor choices included the hardware simulators, which attempted a very high degree of generics (> 50% of code was within a generic) which suffered from severe performance penalties and lack of flexibility in dealing with specific hardware behaviors.

Coding to a common source template is actually a low-tech form of reuse that should not be overlooked. It worked very easily (as long as the template was correct) and served to promulgate good examples for coding and error-handling. Templates were used for declaring, sending and receiving message objects. They worked well, until limitations in the templates were found. A more extensive effort in developing the template would have paid off handsomely in our experience.

Avoid "Monolithic" Ada packages. Trying to be all things to all people will most likely be nothing to anybody. Thinking that object-oriented design translates into "throw everything into one package" is similarly misguided. Use a layered approach instead. Define a package with just type definitions. Then define a package that provides basic operations on these types. Define higher-level packages as necessary to define more complex operations, building on lower-level packages. A careful architecture like this can help you reap big reuse benefits as new uses are found.

Following this approach allows different programs to access the object at different layers of abstraction. Some just need a type definition. Others need basic routines to manipulate the types. Some need advanced routines composed out of basic routines. Others could benefit from automatic initialization of objects at elaboration time (tends to be very trouble-prone, should be carefully controlled by a standards committee). All uses of a complex object, especially potential future ones, may suffer if the only view presented is a single complete monolithic view. A program wishing access to a type definition ends up with pages of "hidden", unused code and data, and maybe even automatic creation/initialization of objects at startup time, referencing databases defined on one computer and not others.

Variable-length strings were another good reusable package. We implemented them with a generic package, pre-instantiated sized to 256 characters. Use of a pre-instantiated package allowed easier sharing of types. However, this also encouraged
waste (programmers were encouraged to use 256-byte strings where only 16 characters were necessary).

Ada Architecture issues

Error handling was our number-one architecture problem. We definitely could have benefitted from better up-front design and more prototyping. Ada tasks complicated the error handling picture drastically.

There is a lot of functional overlap between the capabilities provided by Ada exceptions and those provided by VAX/VMS Condition handlers. There were points of interference or undesirable interplay between the two as well. You need to design error handling into all system service calls. Know which exceptions are worth handling, and which you WANT to be unhandled (because they show up obvious coding or environment problems).

Taking advantage of the operating system's capabilities for calling stack traces on unhandled exceptions, for example, can provide lots of power for debugging. These are especially useful if integrated into the debugging environment, as is the case with most DEC/VAX software.

Concurrency

Ada Tasks

Much fear was generated during early design phases concerning the trade-offs between concurrent operating system processes, and concurrent Ada tasks. During implementation, use was made of both single and multiprocessor machines, with varying results. Software testing and modification history have allowed us to construct better guidelines for process versus task trade-offs. In many cases, processes were used as an aid to work breakdown rather than based on strong architectural need. In some cases these choices caused problems later, and limited the range of available solutions for requirement or design changes. Ada tasking would have been more flexible.

However, increased use of Ada tasking would have required a different development support structure. This support structure would have had to allow separate development and testing of task-based functional work packages independently. The tasks could then be integrated into a single process resulting in a more reliable system.

In general, tasks were well-used and caused relatively few problems. Among the problems were prioritization, blocking, proliferation of tasks required to synchronize between other tasks, and increased rigor in defining/testing the tasking architecture. Tokens (Ada "private" objects containing pointers and flags used in the interface packages between application and service layers) were used to define message addresses. These later became a problem since they were not designed to be shared, yet were shared in some application programs among various tasks. The sheer number of tokens used in the system prevented us from embedding a
task within all token types for synchronization (because of the amount of memory used for task stacks, etc.), but we later embedded "token in-use" flags to help detect instances of sharing. Earlier recognition of the problem would have allowed a range of more elegant solutions.

The following are some additional observations regarding Ada tasks:

- Task context switches are a LOT faster than process context switches. If you're thinking of adding more processes, tasks are better. However, processes are easier to split up the work among multiple independent programmers. Tasks in the same process require more programmer coordination during development.

- Tasks are like lawyers. If you have no tasks, you probably won't need any. However, once you have two tasks, you will probably need another five or ten more to handle coordination between those two tasks plus synchronize any shared inputs, outputs, resources, etc. This means that if you start out thinking that you'll write a program with a few tasks, you'll probably end up writing lots. However, this didn't appear to have been a problem. The number of tasks did not affect performance as long as they were event-driven. You may have to spend more time maintaining relative priorities of tasks as the number increases.

- We avoided PRAGMA TIME_SLICE, since we understood it to add significant overhead. We were successful in avoiding it. Several times we were tempted to use it to alleviate other tasking problems, but it was never absolutely necessary and in the end was successfully avoided.

- Multiprocessor problems were encountered, which required us to use PRAGMA SHARED and PRAGMA VOLATILE, which are implementation dependent. These relied on architecture-dependent features of VMS processors. The features worked well in our two-CPU environment.

- We would have liked to prioritize different entry points in the same task (e.g. to handle the same type of rendezvous, but from different sources), but Ada doesn't allow it. We found a kludgy way of doing it. Instead of attempting reuse, we should have duplicated the task code (i.e. via task types) and prioritized them differently. Maybe we did this because we were attempting excessive reuse, or we were afraid of proliferating tasks. Simpler would have been better.

- We worried a lot about "fairness" of tasking, however all fears appeared to be groundless. If you're worried about fairness of tasking, what you really may be worried about is that you need more CPU power. Or you may have tasks polling when instead you need to turn them around into an interrupt-driven approach.

- Beware of non-reentrant servers, services, etc. Accesses to Rdb, the relational database we used, had to be serialized by routing all task's requests through a single Rdb server task (gateway) which in turn provided
the sole control of the Rdb server. This is a fairly common problem interfacing with non-Ada facilities for which you should watch. Our COTS Graphical User Interface (GUI) non-reentrancy problem was solved with the opposite approach. We ran four copies of it, one for each operator window.

- There was still some question for us about what Delay 0.0 really did, or if it was necessary. It was documented as a way to break the execution of a long-running task and allow a context switch to another waiting task. When we attempted to verify this behavior through benchmarks however, we met with mixed results. We eventually opted not to use the feature. Instead we broke problematic long-running tasks into multiple shorter tasks.

We also had reports of problems with the fairness of allocation of CPU time among tasks. When we investigated with benchmarks, however, all we found were problems with the benchmarks. For each case of purported problems with Delay 0.0 and tasking fairness, programmers who thought they had a problem with an Ada feature were instead using too much CPU time. The ultimate fix was to rearchitect the program to respond to events or Asynchronous System Traps (ASTs) rather than poll.

Compile-time vs. Run-Time Binding

- You can use unchecked_conversion to convert between system.address and object_access types. You’d better be very careful when using this, though. A LOT of errors were committed in this area. Need careful code review and on-the-job indoctrination, perhaps through programmer peer group inspections/walkthroughs, etc. Watch out for things like unintentionally overlayed objects and other C code type pointer errors.

- Anytime you use access types or system addresses in variables it opens the door for memory leaks around allocation/deallocation.

- The Ada compile-time binding of record types was an early problem when data logging record types were very volatile. Many low-worth recompilations were performed. Configuration management and test computer system performance were impacted by the need to accept the many new executables images that were generated. A run-time-binding architecture might have been better in these highly volatile report-writing cases. Once the formats stabilized, the structure did provide for ease of checking. Compilation tests for code impact to changing interface or record format become both routine and precise.

Message Passing Architectures

Ada Interface Definitions

Internal interface definitions, between computers and software subsystems, were captured in Ada. In most cases, representation clauses were not used. Instead the message record definition code was reused in each subsystem. Software configuration management mechanisms
ensured that interfaces were modified consistently. This was reliable since all computers used the same hardware architecture and the same compiler.

**Platform Dependencies**

**Operating System Dependencies**

Many unknown, unforeseen platform dependencies cropped up during the development and test phases. In many cases, these problems were the most astounding and difficult to predict of any we encountered. There is a high degree of functional overlap between the Ada compiler/language run time environment (VAX/VMS Ada 2.2–41 at this writing) and the host/target operating system (VMS 5.5–1). This overlap caused problems in error handling; Ada exception handling interfered with the generation of otherwise automatic operating system calling-tree tracebacks. It also appeared in process management (computer operators couldn’t reliably cancel processes with some types of tasking structures), and debugging (generics and tasks increased difficulty of source-level debugging and thus were unpopular with programmers). While many of these are platform-dependent, they point to the overall problem of overlap between Ada’s functionality and the functionality of the operating system upon which it’s running. If you’re running on a bare-bones processor, or a primitive operating system, then there may be little or no problem. Using a sophisticated and feature–rich operating system like VAX/VMS, on the other hand, can lead to limitations and unforeseen problems when you use Ada’s advanced features and the operating system’s advanced features in the same program.

We ended up having our DEC consultants write a sophisticated assembler routine embedded in each executable which detects unhandled exceptions in any task, forces a traceback, and terminates the image. This has provided us with vastly improved turn–around time for fixing fatal errors found during testing.

Some particulars we found:

- The VAX Ada Run Time Library disables certain features of VMS (like the capability of a computer operator to stop a process gracefully, unless you’ve coded–in your own user–defined exception handler and a means to signal termination). Also, VAX Ada’s memory deallocation/stack unwind during exception propagation interfere with VMS’s capability to do a call tree traceback, which would otherwise have shown a stack dump from the line raising the problem all the way back up to the top of the program. This was especially troublesome when some tasks failed due to unhandled exceptions, (coding errors), but other tasks and the process as a whole, continued to function, making it difficult to detect and isolate the problem.

- Writing debug or error messages using Put_Line caused a performance problem in real–time processes, when all tasks in the process hang behind an operating system output request.
queued to the disk device. We couldn’t tolerate this in many of our hard-real-time executables, so we converted these into shared memory messages between the real-time processes and a lower priority server process, who performed output on behalf of the real-time processes.

- We used tuned Record Management Services (RMS) Input/Output instead of vanilla Ada TEXT_IO or SEQUENTIAL_IO. This was because of the need for heavy-duty tuning, including buffering control and management. We implemented a Mixed I/O-like capability using discriminated records, where each record in the file contained its own embedded record format identifier. This worked quite well, except when the formats were under early development and changed often. Then backward compatibility of current software and previously archived data files became tedious.

- SHARED images (a sophisticated VMS Feature) would have been good to use in certain areas where reusable code made up almost a Megabyte of each executable image, but the integration with Ada was not smooth. By the time we developed a good working approach we had to abandon it because of the retrofit cost. This might have helped Ada’s performance some, in decreasing the memory required. If it could have been done earlier with benefits amortized over more of the development phase, it would have saved money and time. We had initial misgivings about the ability to debug an installed shared image, which later appeared to have been unfounded.

- VMS has a very nice software pseudo-interrupt capability (Asynchronous System Traps or ASTs). The Ada run-time library uses these to do its own synchronization, and instead converts each application AST into a task rendezvous. As a result, running Ada as a part of a “real” AST such as in a call from a device driver written in another language was a difficult proposition (couldn’t use tasking, perform any I/O, etc.). However, the run time library’s conversion of ASTs to tasks (PRAGMA AST ENTRY) was quite accessible to programmers. Tasks seemed to be quite easy (and even natural) to use for this purpose. This enabled anyone to make use of ASTs, whereas without this we probably would had to restrict their use to an elite group of the most experienced programmers.

- Make use of platform capabilities. Don’t be an Ada zealot, thinking you have to write pure Ada code and duplicating functionality otherwise available more cheaply or efficiently in the operating system (100% code portability wasn’t an issue for us – and it may not be for you either). Examples are character and numeric utilities. Just write good (portable) package specs, and implement the bodies of these in the most efficient manner, even utilizing operating system service calls or non-Ada utility packages. This is especially appropriate on complex instruction set computers (CISC) like the VAX. You can always rewrite
the bodies for each new platform to which you port. That way you've addressed performance, reusibility and reduced risk while making good progress and leveraging the capabilities and strong points of your underlying platform.

**COTS Dependencies and Integration**

During the proposal phase of STGT we identified several areas where Commercial Off-The-Shelf (COTS) software could be used. We then deleted costs based on the difference between developing the application from scratch and the cost of the COTS product. However, the following concerns arose:

- We did not allocate necessary additional costs to continually evaluate and incorporate periodic updates/updates of these COTS products. This turned out to be a big ticket item over the life of STGT.

- Purchase good quality COTS bindings. This is a LOT of work. Availability/maturity of Ada bindings should be a significant discriminator during COTS evaluations (e.g., XWindows/Motif binding problems, Distributed File Service (DFS) bindings, device driver bindings, etc.). As usual, productivity may be gained for many at the expense of hard work by a few, or by the purchase of a proper bindings. Consider the trade-offs.

**Performance**

**Ada Performance Characteristics**

Many performance problems were encountered which required various mitigation approaches. Performance modelling was only as good as the input received (much guess work was necessary early on in the life-cycle). This lead to big surprises and varying types of late changes. Eventually larger CPUs and more memory were purchased.

There appears to be a SERIOUS dichotomy in Ada between coding for performance and coding for what most consider to be a "good" Ada style. "Good" Ada was subject to our interpretation of the current literature and to the lessons developed during prototypes by the Ada Core Team. What might be considered "good" Ada of course will change over time. Examples are:

- The generic string package was pre-instantiated for (discriminated record structures) of 256 bytes. This affords maximum reusability and similarity, but appears to waste memory and disk space due in certain cases to needlessly large structures.

- Proponents of "good" Ada often stress deeply nested procedure calls for modularity and reuse. "Fast" Ada is often relatively flat, with a shallow call depth.

- "Good" Ada makes maximum use of local variables. "Fast" Ada allocates
variables once in package bodies, then carefully reuses them within package procedure and function bodies.


- Good Ada makes minimum use of implementation-dependent PRAGMAs. Fast Ada utilizes some PRAGMAs, e.g., PRAGMA ELABORATE to force elaboration of packages before the routines are called for real-time execution.

- As a result of the apparent quandry between "good" and "fast" Ada, it seems that Ada right out of the object-oriented training book can be quite slow. You either need to allocate a bigger CPU, know very accurately the performance characteristics in advance, or plan on a tuning phase to increase the performance of your code once it's written.

Schedule pressures made us opt for the quickest solutions in most cases, that is, larger CPU's. We had some success in optimizing Ada for performance. In some cases the re-coding or reimplementation of a component saved 50–100% of CPU or Memory resources. In one case it saved a factor of 5X CPU for a compute-intensive satellite orbit prediction function.

Configuration Management

Ada Configuration Management

- Ada dependencies are GRAPHS, most library structures/directory hierarchies are TREES. Therefore, if you lock yourself into a library structure that mimics the Ada dependency structure, you'll be disappointed eventually. We used a simple tree of SHARED code at the top, with CSCIs or subsystems below.

- Sublibraries were used versus the VAX Ada Compilation System (ACS) ENTERED units. This allowed automatic recompilation for dependent units when root units changed. The downside was that massive recompilations were forced when not all dependent libraries (and groups using those libraries) were ready to see the change. An alternative approach might have been to develop a tool for automatically re-entering changed units into dependent libraries. That also could have allowed for library dependencies more complicated than a tree.

- We used separate/duplicate libraries to reflect differing levels of software test maturity. For instance, we had one shared set of libraries in which developed code. We only updated the reused components of that library once a week. People affected by interface changes only had to support (or suffer) changes once a week.

- We could have used hierarchical libraries for test, but the computational requirements were too great. Our development CPU resources were never great enough to compile the same source code multiple times for different hierarchical libraries supporting different test maturities. Consequently
ly all tests were forced to the same maturity – fresh from the programmer.

- We had to write a program to extract a cross-reference containing "where-used" information. ACS did not provide this information.

*Ada Compilation Performance*

We did a LOT of work to improve compilation speed. Some of the things we did were:

- Faster CPUs – went from VAX 8250s (1.5 MIPS) to VAX 6610s (25 MIPS).
- More memory – from 64 to 256 Mb
- Tuning of system quotas, batch queues etc.
- RAM DISK and/or semiconductor disk for shared code Ada library (most critical compilation library)
- Spread I/O over multiple disks to reduce bottlenecks
- We didn’t pursue but maybe should have experimented further with the effects of smaller and larger directory/library sizes on compilation speed.

*Ada Compiler*

- We found relatively few bugs. Most were in code generation, a few for floating point types and others which optimized away variables or code. One involved different Ada library unit interfaces depending on whether code was compiled in debug or non-debug. All were resolved in quite good order by excellent DEC support. The lesson was that compiler maturity (for VAX/VMS Ada) was not a risk factor. We also learned that run-time (vs. compile-time) bindings for certain rapidly and persistantly changing functions would have been a much better design from an operational and CM point of view.

- On the other hand, the maturity of ACS was less evident. We have had numerous problems and "features". A good Ada Program Support Environment would be greatly appreciated. We wrote 30,000 lines of "tool" and configuration management scripts. This is significantly more than we anticipated supporting. A good COTS tool available in a timely manner would have been a big productivity enhancement.

- The design of our parent libraries and sublibraries were important. We found ourselves re-creating libraries because library parent/child relationships were hard-coded rather than logical. We redid all libraries with PSEUDO-DEVICE logicals so that successive changes were less painful.

*Project Management*

Equally as important as the Ada lessons learned were the lessons we learned in managing and controlling a large Ada software development effort. Some of these lessons are:

*Standards*

- Our Software Standards and Practices Manual (SSPM) was HUGE. Far too big to be understood or enforced.
• Should have made better use of automated standards checkers or pretty-printer tools.

• Should have tailored the Language Sensitive Editor (LSE) more aggressively for our local standards and included more templates.

• Standards should be issued, proven, taught, understood, reviewed, re-proven, and well documented before any code is written.

Architecture and Schedule

• Allocate the Right CSCIs. We changed the allocation of CSCI's early in the development effort. Changes (reallocations) are difficult to make.

• Avoid Early Split into CSCI Production Groups. We set up a Work Breakdown Structure (WBS) and Management structure on day one. Therefore shifting of work from CSCI to CSCI became a continuous struggle. Work overall system architecture first before parochialism sets in. Set up a mechanism to provide for the overall project good at expense of an individual group.

• Avoid the pressure to accelerate schedules. Believe the "Rule of Tens" (errors found in a later phase take 10 times longer to fix). Missed goals can not be made up. Insist on operation's concepts and equipment (mission equipment) designs prior to software designs.

• View interfaces as a "contract" not as a goal. Interfaces that change are painful.

• Understand tools required and decide on their use well in advance of needs. We developed Configuration Management DCL on-the-fly, did not understand the complexities of Ada CM, and shared interface packages (which are a good idea, but caused massive recompiles). Understand and plan the role of tools throughout the whole life-cycle.

• Define and stick to a fixed methodology. We were guilty of making it up as we go. Much of the heritage we had from our Ada Core Team did not scale up into larger development efforts. Tools did not easily transition between phases.

• Do more prototyping – especially for performance. Make performance estimates based on Executed Lines Of Code (ELOCs) from actual prototypes rather than from Lines Of Code (LOCs) written or predicted to be written. Consider living (non_throwaway) prototypes for broadly used "infrastructure" code.

• Use the right language for the right function. We made some changes to use macro assembler in some critical high frequency applications. Device driver type functions were very slow in Ada as was the high use interprocess communication processes.

• Put Some Teeth into allocating and enforcing performance requirements. We allocated only very high level requirements to the CSCIs for CPU and Memory performance. These were not allocated to lower level components and were therefore untestable and unenforceable.
• Do Code Walkthrus – set aside a team to execute. We relied on peer reviews of code. This became a significant schedule pressure on the CSCI who concentrated more on their own efforts then in a thorough review of another CSCIs code.

• Understand and don’t underestimate the entire domain. Understand the performance aspects of the COTS products and prototype their use. Errors in COTS are harder to fix because of 3rd party involvement. Work with COTS can begin earlier since design effort is usually not required. The effects of the operating system and hardware platform are significant, prototyping and an early start is recommended.

• Know what you are buying and where to use it. For Example, Booch components were excellent at improving productivity. However know their performance characteristics before deciding where to use them and other similar COTS software.

• Hire Experts – utilize vendor consultants. On site expertise is the best way to fix problems and to get preferential access to vendor guru’s and other experts. Often you fix problems before they happen, since consultants can help you with that most difficult assessment, determining what it is that you don’t know.

These Lessons Learned represent only a small subset of the potential data that can be gleaned from GE’s experience on STGT. The main lesson to take away from this paper is that the language, platform, COTS products, tools, etc. are just a means to an end and in themselves are responsible for neither success nor failure.
STGT Ada Lessons Learned

NASA Goddard Software Engineering Laboratory

Software Engineering Workshop

STGT Project
Ada Lessons Learned

Tod Keshri
Bill Manley
Scott Brown
Brian Bauman
Paul Usavage
Don Nagurney

STGT Ada Lessons Learned

Agenda

- Project Overview
- Software Configuration
- Software Metrics
- Ada Project Management Lessons Learned
  - Project Schedule/Structure
  - General Issues
  - Performance/Sizing
  - Reusability
- Ada Lessons Learned
  - Generics
  - Tasking
  - COTS/Platform Dependencies
  - Package Structuring/Record Formats
  - Exceptions

SEL-92-004 page 3
STGT Ada Lessons Learned
Project Overview

STGT Second TDRSS
Ground Terminal
Dec 2-3, 1992

GE DELIVERABLES
- 3 - 19M S/Ka-BAND ANTENNAS
- 175 RACKS OF GROUND COMM EQUIPMENT
  - DATA RATES 1Kbps TO 300 Mbps
- 250 ADPE RACKS
- 500K LINES Ada SOFTWARE

STGT Ada Lessons Learned
Software Configuration

DIS ADPE
DIS CSCI
COM CSCI

SGLT 1 EXEC ADPE
EXC CSCI
COM CSCI

K-band TTC
TTC CSCI
COM CSCI

USS SA1
USS CSCI
COM CSCI

USS SA2
USS CSCI
COM CSCI

USS MA
USS CSCI
COM CSCI

COM CSCI
DIS CSCI
EXC CSCI
USS CSCI
TTC CSCI
WKS CSCI
SIM CSCI
MDS CSCI
Common Services
NCC Interface
SGLT Scheduling
Equipment CMD/MON
TDRS/Antenna Control
Operator Interface
SMTF Simulation
SMTF Tools

Chart 3

Chart 4
### STGT Ada Lessons Learned

#### Software Metrics

<table>
<thead>
<tr>
<th>CSCI</th>
<th>Size (LOC)</th>
<th>Hours</th>
<th>LOC/Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTC</td>
<td>100000</td>
<td>115000</td>
<td>.86</td>
</tr>
<tr>
<td>DIS</td>
<td>76000</td>
<td>79000</td>
<td>.96</td>
</tr>
<tr>
<td>USS</td>
<td>71000</td>
<td>58000</td>
<td>1.22</td>
</tr>
<tr>
<td>EXC</td>
<td>26000</td>
<td>23000</td>
<td>1.13</td>
</tr>
<tr>
<td>WKS</td>
<td>152000</td>
<td>56000</td>
<td>2.7</td>
</tr>
<tr>
<td>COM</td>
<td>23000</td>
<td>37000</td>
<td>.62</td>
</tr>
<tr>
<td>MDS</td>
<td>100000</td>
<td>36000</td>
<td>2.77</td>
</tr>
<tr>
<td>SIM</td>
<td>40000</td>
<td>40000</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Total 588000 444000 1.32

1 - Requirement Analysis thru Software Test
2 - Includes Cost of Common Ground Control/Monitor and Fault Detection
3 - Includes Common Ground Control/Monitor and Fault Detection

### Ada Project Management Lessons Learned

- Project Schedule/Structure
- General Issues
- Performance/Sizing
- Reusability
• Allocate the Right CSCIs
  Changes (Reallocation) are Difficult to Make

• Avoid Early Split into CSCI Production Groups
  Work Overall System Architecture First
  Set up a Mechanism to Provide for the Overall
  Good at Expense of an Individual Group

• Avoid the Pressure to Accelerate Schedule
  Believe the “Rule of Tens”
  Missed Goals Can Not Be Made Up
  Insist on Operation’s Concepts and Equipment Designs Prior to Software Designs

• View Interfaces as a “Contract” not as a Goal
  Interfaces That Change Are Painful

• Understand Tools Required and Decide on their Use Well in Advance of Needs
  CM Developed DCL on-the-fly : did not understand the Complexities of Ada: Shared Interface Packages (A Good Idea) Caused Massive Recompiles
  Understand and Plan the Role of Tools Throughout the Whole Lifecycle

• Understand and Don’t Underestimate the Entire Domain
  COTS
  DEC/VMS

• Prototype and Utilize Prototype Code Everywhere

• Hire the Right People Then Train/Train/Train
STGT Ada Lessons Learned
Ada Project Management
Lessons Learned
General Issues

- Define and Stick to A Fixed Methodology
  Define in Advance and Don't Experiment
  Educate User's
- Keep the SSPM Simple – Useful and Easy to Enforce
- Do Code Walkthrus – Set Aside a Team to Execute

STGT Ada Lessons Learned
Ada Project Management
Lessons Learned
Performance/Sizing

- More Prototyping – Estimates Based on Executed LOCs
- Complex Generics Proved to be Extremely Slow
- Understand Compile and Link Process (e.g. Compiler Eliminates Dead Code But Linker Does Not)
- Use the Right Language for the Right Function
- Bad Ada Is Real Baaaaad
- Put Some Teeth into allocating and enforcing Performance Requirements
- Know What You are Buying and Where to Use It
  Booch Components – Not Optimized for Performance
- Don’t Attempt High Level Generics Yet
  Ground Equipment Simulation Is the Wrong Choice
- Provide for Project Wide Reuse Czar
  Avoid Parochialism
  Proactive Search for Opportunities

- Ada Lessons Learned
  - Generics
  - Tasking
  - COTS/Platform Dependencies
  - Package Structuring/Record Formats
STGT Ada Lessons Learned
Ada Lessons Learned
Generics

- Can be a Performance Problem
- Are to Debug with Interactive Source Level Debugger
- Keep Small: Don't Attempt a Reusable Ground Station
- Restrict Usage to Types as Formal Parameters
- Keep Them out of the Hands of Amateurs
  Limit to Your Most Experienced People
  Review/Review/and Then Again – Prototype Performance

Mistrusted at First – Found Many Appropriate Uses
- Understand the Target Environment/Prototype
- Provide for Terminate Alternatives – Make Sure a Parent can Terminate Children
- Exceptions Must Be Propagated Upward (Free Running Tasks Need Some Control)
- Don't Substitute Tasks Where Procedures Would Suffice
- When Using Tasks – Centralize Control (one writer)
- If You Plan on a Few Expect Many More
STGT Ada Lessons Learned

- Understand Compiler/Linker and Their Interaction
  Don't Count on Default Order of Elaboration
- Understand The Whole Domain
  VMS Services Better Than Ada Features
- Pick COTS With Ada Bindings (Avoid Multiple Translations)
- SQLMODS Proved to Be Workable Interface
  Imbedded SQL was Impossible to Debug
- Hire Experts – Utilize Vendor Consultants
- Product Upgrades are Large Undertakings and Come at the Most Inopportune Times
  Properly Plan for and Fund Product Upgrades
- Avoid the Creation of Processes Without Justification

STGT Ada Lessons Learned

- Limit Scope of Packages – Don't Try to Encapsulate and Entire Object in One Package
  Use Multiple Packages – Each With a Purpose
  Know the Intended Use of the Packages (e.g. Senders vs Receivers)
  Avoid Monoliths
- Don't Put Database Access into Interface Packages
- Don't Combine Loosely Related Types
- Create Null Instances of a Type as an Initial Value
- Avoid String Types – Usually Masking an Enumerated Type
- Renaming – Many Differences of Opinions: Be Careful
- Use Only For Real Errors – Very Expensive for Use As GOTOs
- "When Others" obscures origin of exceptions
- Understand and Plan for Unhandled Exceptions
  Tracebacks and Stack Dumps are Good Debugging Tools
  Process/System Dumps Have Their Place
- Specify and Design Expected Levels of Error Handling

STGT Ada Lessons Learned
Summary

- Project Pressures Force Old Habits to Return
- Solidify Interfaces Under Penalty of Death
- Prototype Everything and Always
- Enforce Performance Allocations
- Focus Reuse and Dedicate Resources
- Restrict Generics
- Don't Be Afraid of Tasks
- Understand the Domain – and Hire Where Necessary
- Limit Scope of Packages
- Be Prepared to Upgrade COTS
Panel: Is Ada Dying?

Marv Zelkowitz, University of Maryland, Facilitator
Stu Feldman, Executive Director of Computer Systems Research, Bellcore
John Foreman, Director of STARS Program, Department of Defense
Susan Murphy, AAS Software Manager, IBM
Tom Velez, President and CEO, CTA
Panel: Is Ada Dying?

- Facilitator:
  - Marvin V. Zelkowitz, NIST/CSL and Department of Computer Science, University of Maryland

- Panelists:
  - Stu Feldman, Executive Director, Computer Systems Research, Bellcore
  - John Foreman, Director of STARS Program, DARPA
  - Susan Murphy, AAS Software Manager, IBM FSC
  - Tom Velez, President and CEO, CTA

SEL interest in Ada

- Why SEL interest in Ada?
  - SEL has broadest experience with Ada within NASA
  - SEL has collected much data on the use of Ada (as well as many other technologies)
  - SEL has analyzed Ada usage from various perspectives (e.g., see last few Workshop proceedings)

- Results of SEL studies:
  - Value of Ada not unconditionally shown
  - Need to assess current status and plan future processes
**SEL Ada Projects**

- **Ada Studies**
  - 1 Parallel Study Completed
  - 9 Ada Production Products Completed
  - All Projects Provide Full SEL Data
  - Numerous Studies Completed

- **NUMBERS**
  - EUVEDSIM 184K
  - EUVETELS 67K
  - UARSTELS 68K
  - FDAS 68K
  - GOESIM 92K
  - GOADA 170K

**Parallel Development – Ada and FORTRAN**

---

**Ada (and OOD) Impacts on Cost**

- **Cost To Develop**
  - Effort per Developed Statement
    - FORTRAN
    - GRODDY
    - Ada

- **Cost To Deliver**
  - Effort per Delivered Statement
    - FORTRAN
    - GRODDY
    - Ada

*Ada: Developed Size = 100% New + 30% Old Statements*

- Development cost per statement has been no cheaper for Ada
- Reuse potential of Ada is significant
- Reuse cost factor has changed in Ada systems

---

SEL-92-004 page 394
But FORTRAN reuse is also growing

Language use in Code 500 at Goddard
NASA IBM mainframe Ada evaluation

- Need more development and testing support
  - Two compilers evaluated
  - Multiple source file compilation limited
  - Ada library can be corrupted
  - Inflexible Ada library manager
  - Need better debugger
  - One compiler failed to even compile some modules

- Need improvement in error handling and error messages
- Need improvement in performance

- Result: Could not use IBM mainframe for large-scale NASA Goddard development

Onboard embedded Ada application

- Goal: Dual 1750A processors with shared memory to handle onboard navigation

- Environment: TI 1750A hardware, Tartan cross compiler system on VAX

- Problems: Intermittent communication and shared memory problems. Hardware and software vendors could not solve problems.

- Resolution: Had to fly uniprocessor system with reduced functionality.
Positive attributes of Ada

- Language syntax and semantics are in mainstream language design – an outgrowth of FORTRAN, ALGOL and Pascal
- Language features to aid in large system design, reuse and maintenance (e.g., packages, tasking, exceptions, generics)
- Over 250 validated compilers
- University use growing – 14 Ada textbooks and use at perhaps 10% of U.S. universities (from: November, 1992 Comm. of the ACM)
- Millions of lines of Ada code for commercial non-military applications – Examples: Shell Oil for exploration, Motorola for cellular telephones, Boeing for 747-400, GE for automated steel manufacturing, NTT (Japan) for commercial telecommunications applications, Nokia SoftPlan (Finland) for a banking system, plus others
- Ada-9X revision to solve many of the lingering problems

Negative attributes of Ada

- Hard to learn to use well
- Lack of production quality compilers
- Performance penalty in certain critical applications
- Doesn’t handle object oriented design – Impact of C++
Observations

After 10 years of development...

- Growth of courses and textbooks in Ada seems very slow.
- Does not seem to be a large scale movement to Ada within non-DoD segments of the industry. Most examples are anecdotal.
- Ada does not yet seem ready within the large mainframe environment at Goddard.
- Yet, seems to be a natural attraction to C and C++. Both have attained huge unsupported growth.

Will there be supported Ada products in 10 years?

Summary of issues

- "Many of the perceived problems with Ada were due to the immaturity of early implementations, rather than flaws of the language itself. Some of these perceptions linger, even though mature Ada implementations are available today and most of the previously identified shortcomings have disappeared." – Erhard Ploedereder, Comm. of the ACM, Nov., 1992

- Is Ada today an economically viable language for building software systems?
- If so, for what class of projects is it appropriate?
- If not, what criteria are needed for determining the economic viability of Ada (and when should they be met)
Panel organization

• Opening statements:
  - What is your position and why?
  - What are the objective or subjective criteria supporting your position?
  - What actions should the principles be taking (i.e., DoD, NASA, contractors) and what will Ada be in the next century?

• Each panelist will talk for up to 10 minutes; then a 5 minute comment by panelists on other statements; then general comments or questions from workshop attendees
Uses and Future

<table>
<thead>
<tr>
<th>Niche</th>
<th>1980</th>
<th>=&gt; 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial</td>
<td>COBOL</td>
<td>+4GL</td>
</tr>
<tr>
<td>Scientific/Engineering Systems</td>
<td>FORTRAN</td>
<td>+C</td>
</tr>
<tr>
<td></td>
<td>ASM, C</td>
<td>C</td>
</tr>
<tr>
<td>Prototyping</td>
<td>LISP, SMALLTALK</td>
<td>C, PROLOG ...</td>
</tr>
<tr>
<td>Embedded/Real Time</td>
<td>ASM, ADA</td>
<td>C, ADA</td>
</tr>
<tr>
<td>S/W Engineering</td>
<td>ADA</td>
<td>C++, ADA, ...?</td>
</tr>
<tr>
<td>CS Research</td>
<td>C, LISP</td>
<td>C++, CLOS, ML</td>
</tr>
</tbody>
</table>

Sociology

Lifecycle
- Born/Stillborn
- Born Again?

Nurture
- Phoenix/Bride of Frankenstein?

Kinship
- None Allowed

Support System
- Ada Industry $= \frac{d^n}{dt^n}$ Defense Budget

Ecology
- Niches and Competition
Unproven Comparisons

Software Maintainability

<table>
<thead>
<tr>
<th>Ada &gt; C</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada &gt; C++</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Language Complexity

Ada 9X > FORTRAN 90 > C++ > C

Simple - Compiler Difficulty

Ada 9X > Ada > C++ > C

Excellent - Compiler Difficulty

C++ > C > Ada > FORTRAN

Ada Properties

+ Complete
+ Supported
+ Sponsored
+ Real-Time
+ Software-Engineering
? Configuration Support

- Syntax
- Garbage Collection
- Complexity
- Software Support
- Use in Systems ("open")
- Love
IS ADA DYING?

John Foreman
DARPA/SISTO
(703) 243-8655
jff@sei.cmu.edu

POSITION

- NOT dying, generally in good shape
  - Still maturing
  - Still potential for growth
  - Real tech insertion and transfer takes long time
  - Is the receptor community mature?
  - Too much 'over expectation'
  - DoD still has unique requirements to satisfy
CRITERIA FOR JUDGEMENT

- Tool quality continually better
- HW base much improved (32 bit processors, etc)
- Real projects/real results
- Use of language for large projects
- Overseas use
- Stability and validation are important

GETTING TO THE YEAR 2000

- Planned 9X insertion and use (bindings)
- Case studies
- Do something about people: education
- Need changes to acquisition process
  - life-cycle perspective
  - incremental builds
  - product evolution
- Process/product considerations
- Software product line management
  - software architectures
  - COTS
- Consider effects of downsizing
  - niche market
  - polylingualism
"IS ADA DYING?"

SUSAN MURPHY
AAS SOFTWARE FUNCTIONAL MANAGER
DECEMBER 3, 1992

ADA
IS
ALIVE AND WELL
ON THE
FAA'S ADVANCED AUTOMATION SYSTEM (AAS)
### AAS Program Highlights

Over 2.5 million lines of newly developed code (mostly ADA)

<table>
<thead>
<tr>
<th>Segment</th>
<th>KSLOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Sector Suite System (ISSS)</td>
<td>1058</td>
</tr>
<tr>
<td>Terminal Advanced Automation System (TAAS)</td>
<td>716</td>
</tr>
<tr>
<td>Tower Control Computer Complex (TCCC)</td>
<td>257</td>
</tr>
<tr>
<td>Area Control Computer Complex (ACCC)</td>
<td>448</td>
</tr>
</tbody>
</table>

**Four Segments**

By year 2000, AAS segments will be in use throughout the USA and for foreseeable future.

- 432 Towers
- 186 Terminals (TRACON)
- 23 Enroute Centers (ARTCC)

Many hundreds of ADA programmers involved with AAS over life of the program.

AAS is basis of worldwide ATC programs/BIJS

- Republic of China (Taiwan)
- U.K.'s New Enroute Center (NERC)
- Germany
- Sweden
- EUROCONTROL (ODS)
- Mexico
- Belgium
FOR ADA TO GROW:

ADA 9X MUST BE FULLY DOWNWARD COMPATIBLE WITH ADA 83
(NO CODING CHANGES REQUIRED)

ELSE

- THESE PRODUCTION SYSTEMS WILL NOT TRANSITION TO ADA 9X
- HUNDREDS OF ADA PROGRAMMERS WILL NOT EVOLVE TO USE OF
ADA 9X FEATURES
AIR FORCE ADA PROJECTION

MAJOR TOM CROAK, USAF

1991 SURVEY* 1991 PROJECTION

<table>
<thead>
<tr>
<th>Language</th>
<th>Percentage</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>40%</td>
<td>20%</td>
</tr>
<tr>
<td>ADA</td>
<td>10%</td>
<td>40%</td>
</tr>
<tr>
<td>FORTRAN/JOVIAl</td>
<td>30%</td>
<td>25%</td>
</tr>
<tr>
<td>C</td>
<td>3%</td>
<td>10%</td>
</tr>
<tr>
<td>OTHER</td>
<td>17% (450 LANG'S.)</td>
<td>5% (250 LANG'S.)</td>
</tr>
</tbody>
</table>

THERE HAVE BEEN NO ADA WAIVERS SINCE JULY 1990

*ALL OPERATIONAL SYSTEMS; ADDITIONAL 32M OF ADA CODE UNDER DEVELOPMENT

ADA INFORMATION CLEARINGHOUSE

ADA PROJECTS

- ACADEMIA 4
- ARMY 62
- NAVY 220
- MARINE CORPS 41
- AIR FORCE 151
- COMMERCIAL 111
- GOV'T. (NON-DoD) 58
- INTERNATIONAL 68
- OTHER DoD 7

TOTAL 722

EXAMPLES

"SUB-SIM" ATTACK SUB SIMULATOR
ADVANCE FIELD ARTILLERY TACTICAL DATA SET (AFATDS)
ADVANCE SURVEILLANCE WORKSTATION
NAVAL FLIGHT RECORD SUBSYSTEM
ADVANCED TACTICAL FIGHTER (F22)
BOEING 777
ADVANCED AUTOMATION SYSTEM (AAS)
NETHERLANDS TELEPHONE CONTROL & MONITORING SYSTEM
SINGLE CHANNEL OBJECTIVE TACTICAL TERMINAL (SCOTT)
ADA & C++ - BUSINESS CASE ANALYSIS*

ADA

28 COMPANIES W/VALIDATED PRODUCTS

GOVT. CONTROLLED/ANSI & ISO STANDARDS

YES

22 UNIVERSITIES & 13 DoD INSTALLATIONS

78.8

210 (SLOC/MM) (153 DATA POINTS)

65 (SLOC/MM) (153 DATA POINTS)

24 (153 DATA PTS.)

1 (153 DATA PTS)

1631 (23% HIGHER)

1738 (24% HIGHER)

C++

MARKET AVAILABILITY

18 VENDORS OFFER C++

STRONG STANDARDIZATION

NO VALIDATION OR STANDARD EXIST

CROSS COMPILATION

NO

EDUCATION/TRAINING

4 UNIVERSITIES

FEATURE COMPARISON** (OUT OF 100)

63.9

PRODUCTIVITY

107 (SLOC/MM) (38 DATA PTS.)

(COMP: 183 ALL LANG.)

COST

55 (23 DATA PTS.)

(NORM: 70 ALL LANGUAGES)

AVG. ERROR RATES (PER KSLC)

31 (23 DATA PTS.)

INTEGRATION

(3: NORM ALL LANGUAGES)

FORMAL QUAL TEST

3 (23 DATA PTS.)

(3: NORM ALL LANGUAGES)

ADA COCOMO COST ANALYSIS

MIS

1324

C3 SYSTEMS

1401

* BASED ON U.S. AIR FORCE STUDY
** BY SEI FOR APPLICATIONS INFORMATION/C3 SYSTEMS
<table>
<thead>
<tr>
<th>OBJECTIVE</th>
<th>RESULT</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE (DoD-1) HOL</td>
<td>WE (CTA) SEE ADA MANDATED IN VIRTUALLY 100% OF DoD RFPs</td>
<td>+</td>
</tr>
<tr>
<td>SUPPORT MODERN SOFTWARE ENGINEERING TECHNIQUES</td>
<td>YES: THROUGH STRONG TYPING PACKAGING, AND OTHER FEATURES</td>
<td>+</td>
</tr>
<tr>
<td>PROVIDE AN &quot;ADA&quot; ORIENTED PROGRAMMING ENVIRONMENT</td>
<td>NO: CLEARLY, THE PROMISES OF CAIS, APSE, NOT REALIZED</td>
<td>-</td>
</tr>
<tr>
<td>INCREASE OF PRODUCTIVITY</td>
<td>NO CLEAR, CONCLUSIVE RESULTS - APPARENT RESULT IS SAME AS OTHER LANGUAGES</td>
<td>NEUTRAL</td>
</tr>
<tr>
<td>DECREASE LC SOFTWARE MAINTENANCE (EVOLUTION) COST</td>
<td>EVIDENCE IS POSITIVE - LESS ERRORS IN O&amp;M</td>
<td>+</td>
</tr>
<tr>
<td>STANDARDIZATION</td>
<td>YES: ANSI &amp; ISO</td>
<td>+</td>
</tr>
<tr>
<td>CONTROLLED, STABLE COMPILER IMPLEMENTATION</td>
<td>YES: THROUGH GOVT. SUPPORT</td>
<td>+</td>
</tr>
<tr>
<td>CLEAR &quot;GRASS ROOTS&quot; USAGE (IN COMMERCE, ACADEMIA)</td>
<td>NO: CERTAINLY NOT LIKE &quot;C&quot;</td>
<td>-</td>
</tr>
</tbody>
</table>

OVERALL RESULT: POSITIVE
# Appendix A: Attendees

| Abd-El-Hafiz, Salwa K., University of Maryland |
| Addelston, Jonathan D., Planning Research Corp. |
| Agresti, Bill W., MITRE Corp. |
| Aikens, Stephen D., DoD |
| Allen, Julia, Software Engineering Institute |
| Allen, Russ, IRS |
| Anderman, Al, Rockwell SSD |
| Anderson, Barbara, Jet Propulsion Lab |
| Anderson, Jim, IRS |
| Angier, Bruce, Institute for Defense Analyses |
| Arnold, Robert S., Sevtec |
| Astill, Pat, Centel Federal Services |
| Austin, James L., IRS |
| Ayers, Everett, Ayers Associates |
| Bachman, Scott, DoD |
| Bacon, Beverly, Computer Sciences Corp. |
| Bailey, Carmine M., McDonnell Douglas |
| Bailey, John, SEL |
| Balick, Glenn, DoD |
| Barbara, Edward K., U.S. Air Force |
| Barbour, Ed, U.S. Air Force |
| Barnes, Bruce H., National Science Foundation |
| Barnette, Randy, Hughes STX |
| Barnhart, Don, Boeing Aerospace Co. |
| Basch, Bill, Boeing Computer Support Services Co. |
| Basili, Vic, University of Maryland |
| Bates, Bob G., Lockheed Space Operations |
| Baumert, John H., Computer Sciences Corp. |
| Bearcell, Deborah J., Computer Sciences Corp. |
| Beatty, Kristin, IIT Research Institute |
| Belle, Jeffery C., Logicon, Inc. |
| Beswick, Charlie A., Jet Propulsion Lab |
| Billick, Ron, Bell Atlantic |
| Binegar, Scott, Computer Sciences Corp. |
| Biondi, Marisa, IRS |
| Bishop, Steven, Naval Air Warfare Center |
| Bisignani, Margaret, MITRE Corp. |
| Bissonette, Michele, Computer Sciences Corp. |
| Blackwelder, Jim, Naval Surface Warfare Center |
| Biagmon, Lowell E., Naval Center for Cost Analysis |
| Blankenship, Donald D., U.S. Air Force |
| Blankenship, Gordon, U.S. Air Force |
| Bloodgood, Pete, IRS |
| Blough, Lyn, Computer Sciences Corp. |
| Blum, Bruce I., Applied Physics Lab |
| Bogdan, Robert J., Computer Sciences Corp. |
| Bojerg, Jacqueline, Computer Sciences Corp. |
| Boland, Dillard, Computer Sciences Corp. |
| Bond, Jack, DoD |
| Boon, Dave, Computer Sciences Corp. |
| Booth, Eric, Computer Sciences Corp. |
| Borger, Mark W., Software Engineering Institute |
| Boyce, Glenn W., MITRE Corp. |
| Bozenski, Richard, DoD |
| Bozoki, George J., Lockheed |
| Bradley, Stephen, MMS Systems |
| Bradshaw, Royce, NATO |
| Brandt, Thomas C., Software Engineering Institute |
| Bredeson, Mim, Space Telescope Science Institute |
| Briand, Lionel, University of Maryland |
| Brill, Gary, IRS |
| Brisco, Phil C., Hughes STX |
| Brown, Robert E., Hughes Aircraft Co. |
| Brownsword, Lisa L., Computer Sciences Corp. |
| Brownsword, Robert J., Rational |
| Bruhn, Anna, Jet Propulsion Lab |
| Bullock, Steve, IBM |
| Bunch, Aleda, Social Security Administration |
| Buresl, Billie, IBM |
| Burns, Patricia, Computer Sciences Corp. |
| Butler, Sheldon, Computer Sciences Corp. |
| Butterworth, Paul, Hughes STX |
| Button, Janice, DoD |
| Button, Judee, IRS |
| Caldiera, Gianluigi, University of Maryland |
| Calvo, Robert, Paramax Aerospace Systems |
| Cantalupo, Joy, IIT Research Institute |
| Capraro, Gerald T., Karman Sciences |
| Card, Dave, Computer Sciences Corp. |
| Carlin, Catherine M., Dept. of Veterans Affairs |
| Carlisle, Candace, NASA/GSFC |

**PRECEDING PAGE BLANK NOT FILMED**
Manter, Keith, Computer Sciences Corp.
Marciniak, John, Computer Technology Associates, Inc.
Marcoux, Darwin, DoD
Marquiss, Terri, Computer Sciences Corp.
Martin, Carol, TRW
Mashiko, Yasuhiro, University of Maryland
Mauney, Mike, Census Bureau
Maury, Jesse, Omitron, Inc.
McConnel, Pat, IRS
McCreary, Julia M., IRS
McGarry, Frank E., NASA/GSFC
McGarry, Mary Ann, IIT Research Institute
McGarry, Peter, General Electric
McGovern, Dan, FAA
McKay, Judith A., Census Bureau
McKinney, Cathy, IRS
McKinney, Jimmie, USAFIS
McNeill, Justin F., Jet Propulsion Lab
McSharry, Maureen, Computer Sciences Corp.
Mehta, Shilpa, American Systems Corp.
Meick, Douglas, Library of Congress
Mendonca, Maaoel G., University of Maryland
Merry, Paul, Harris Space Systems Corp.
Miller, Ronald W., NASA/GSFC
Miller, Sharon E., AT&T Bell Lab
Miller, Terrence, Project Engineering, Inc.
Mills, John P., Booz, Allen & Hamilton, Inc.
Mohallatee, Michael, Computer Sciences Corp.
Moleski, Laura, CRM
Moleski, Walt, NASA/GSFC
Moniuszko, Charles, DoD
Moore, Betty, IRS
Moore, Kathryn J., USAISSDC-A
Moore, Paula, NOAA/SPOx3
Moortgat, Jean-Jacques, Booz, Allen & Hamilton, Inc.
Morasca, Sandro, University of Maryland
Morgan, Elizabeth, Bendix Field Engineering Corp.
Morusiewicz, Linda M., Computer Sciences Corp.
Mostoller, Brad, Sunquest Information Systems
Moxley, Fred L., DISA/CFS
Mucha, John F., IRS
Muckel, Jerry, Computer Sciences Corp.
Mulville, Daniel, NASA/HQ
Munkeby, Steve, Martin Marietta
Murphy, Susan, IBM
Murtha, Kimberly N., Sunquest Information Systems
Myers, Philip I., Computer Sciences Corp.
Myers, Robert M., MITRE Corp.
Narrow, Bernie, Bendix Field Engineering Corp.
Nassau, Dave, Applied Expertise, Inc.
Newman, Phillip A., NASA/GSFC
Nishimoto, Theresa, Coopers & Lybrand
Nola, Charles L., NASA/MSFC
Noonan, Carolina, Computer Sciences Corp.
Noone, Estelle, Computer Sciences Corp.
Norcio, Tony F., University of Maryland Baltimore County
O'Brien, Robert L., Paramax Aerospace Systems
O'Connor, Sean, Martin Marietta
O'Neil, Don
O'Neil, Patrick, U.S. Army AMSAA
O'Neil, Peter, PA Blue Shield
Olmacher, Jane A., Social Security Administration
Okupski, Scott, U.S. Air Force
Olson, Lenoard, Hughes STX
Osifchin, Tammy, Hughes Aircraft Co.

Padgett, Kathy, Census Bureau
Page, Gerald, Computer Sciences Corp.
Pajerski, Rose, NASA/GSFC
Palmer, Regina, Martin Marietta
Paluzzi, Paul, Computer Sciences Corp.
Pang, Les, FAA
Panilillo-Yap, Nikki M., IBM Canada Ltd.
Park, Robert, Computer Sciences Corp.
Patrick, Debora, IIT Research Institute
Patterson, F. G., NASA SSFO
Patton, Kay, Computer Sciences Corp.
Pavonica, Paul, Treasury/Fincen
Pecore, Joseph N., Vitro Corp.
Peeples, Ron L., IBM
Pendergrass, Vicki, NASA/GSFC
Pendley, Rex, Computer Sciences Corp.
Peng, Wendy, NIST
Peng, Yuh-Fen, Computer Sciences Corp.
Perry, Brendan, Hughes STX
Peters, Jeffrey, U.S. Air Force
Petitjohn, Margot, IRS
Philipot, Donn E., Technology Applications, Inc.
Sorensen, Steven, Martin Marietta
Sova, Don, NASA/HQ
Spangler, Alan R., IBM
Spence, Bailey, Computer Sciences Corp.
Spencer, Mike, Naval Air Warfare Center
Spool, Peter R., Siemens Corporate Research, Inc.
Sporn, Patricia A., NASA/HQ
Squires, Burton E., Mnemonic Systems Inc.
Srivastava, Alok, Computer Sciences Corp.
Stanton, Faye, IRS
Stark, Michael, NASA/GSFC
Stauffer, Mike P., General Electric
Stevens, Jan, Systems Research & Applications Corp.
Stewart, Barbara C., U.S. Air Force
Strano, Caroline, FAA
Sugumaran, Vijayan, George Mason University
Svara, Allan C., USAF/7th, Comm, Group
Swain, Barbara, University of Maryland
Szulewski, Paul S., Draper Labs, Inc.
Tasaki, Keiji, NASA/GSFC
Tausworthe, Robert C., NASA/JPL
Tavakoli-Shirajl, Iraj, George Mason University
Tervo, Betsy, Computer Sciences Corp.
Thackrey, Kent, Planning Analysis Corp.
Theeke, Patrick, Electronic Warfare Associates, Inc.
Theofanos, Mary, Martin Marietta
Thomas, Donna C., Computer Sciences Corp.
Thomas, Isac, Computer Sciences Corp.
Thomas, William, MITRE Corp.
Thomen, Mark, IBM
Thrasybule, Wesner, Computer Sciences Corp.
Tipparaju, Suri, Hughes STX
Tisnado, Gilberto M.
Tran, Dennis A., MITRE Corp.
Tran, Tuyet-Lan, Jet Propulsion Lab
Trujillo, Nelson W., NDU/IRM
Truong, Son, NASA/GSFC
Tsagos, Dinos
Tupman, Jack R., Jet Propulsion Lab
Ullman, Richard, Hughes ST Systems Corp.
Usavage, Paul, General Electric
Valett, Jon, NASA/GSFC
Valleni, Bob R., TRW
Van Meter, David, Logicon, Inc.
Van Verth, Patricia E., Canisius College
VanHorn, Wendy J., IRS
Varklett, Vanessa, IIT Research Institute
Vaughan, Joe, Social Security Administration
Vause, David G., IBM
Vazquez, Federico, Computer Sciences Corp.
Velez, Tom E., Computer Technology Associates, Inc.
Verducci, Anthony J., AT&T Bell Lab
Viola, Ken W., IRS
Voit, Eric, Bell Atlantic
Votta, Lawrence G., AT&T Bell Lab
Wagoner, Raelene, Systems Research & Applications Corp.
Walligora, Sharon R., Computer Sciences Corp.
Wallace, Dolores, NIST
Walsh, Bob, IRS
Walsh, Chuck, NASA Center for Aerospace Information
Waszkiewicz, Mary Lily, Computer Sciences Corp.
Weber, Paul A., Technology Planning, Inc.
Weiss, Peter, Arthur D., Little, Inc.
Weiss, Sandy L., GTE
Wells, Robert, Computer Sciences Corp.
Werling, Richard, Software Productivity Consortium
Wessale, William, CAE-Link Corp.
Weston, William, NASA/GSFC
Weszka, Joan, IBM
Wheeler, J. L., Computer Sciences Corp.
White, Cora P., New Technology, Inc.
Whitehead, John W., NAVSEA 06D3
Whitfield, Josette, IIT Research Institute
Whitman, Cynthia B., USAISDC-A
Wilkins, Elsie C., USAFISA
Williamson, Jim, Sunquest Information Systems
Wilson, Jim, Applied Expertise, Inc.
Wilson, Randy D., Naval Center For Cost Analysis
Wingfield, Lawrence D., Computer Sciences Corp.
Wisdom, Rex, U.S. Air Force
Wise, Charles F., Technology Applications, Inc.
Wong, Sha, IMS
Wong, Yee, Computer Sciences Corp.
Wood, Richard, Computer Sciences Corp.
Wood, Terri, NASA/GSFC
Woodward, Herbert P., TRW
Worley, Patricia W., Boeing Computer Support Services Co.
Wortman, Kristin, Hughes STX

Yin, Sandra, IRS
Youman, Charles, SETA Corp.
Young, Andy, Bendix Field Engineering Corp.
Yu, Anna, NC A&T State University

Zavaleta, Henry M., Computer Sciences Corp.
Zaveler, Saul, U.S. Air Force
Zelkowitz, Marv, University of Maryland
Zimet, Beth, Computer Sciences Corp.
Zucconi, Lin, Lawrence Livermore National Laboratory
Zvegintzov, Nicholas, Software Maintenance News Inc.
Appendix B: Standard Bibliography of SEL Literature
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-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978


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


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-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-82-004, *Collected Software Engineering Papers: Volume 1*, July 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-82-1106, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1992


BI-2


BI-3


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


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


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


SEL-RELATED LITERATURE


BI-6


BI-10


BI-11


NOTES:

0This document superseded by revised document.

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

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

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

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

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

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

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

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

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

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