

Appendix I

Software Support

Content

I1.0 Tab 1: The DoD Generic Fighter: F-22's Historical Foundation	I-4
I1.1 Introduction	I-4
I1.2 Generic Fighter Architecture	I-4
I1.2.1 Variants	I-5
I1.3 Software Changes	I-7
I1.4 Software Support Process	I-9
I1.4.1 Support Costs	I-11
I1.4.2 Support Strategy	I-12
I1.4.3 Support Environment	I-13
I1.5 Training System Impacts	I-14
I1.6 Program Management	I-14
I1.7 Lessons Learned	I-14
I1.8 Acknowledgments	I-15
I2.0 Tab 2: COTS Integration and Support Model	I-16
I2.1 Abstract	I-16
I2.1.1 Observations	I-16
I2.2 Program Model	I-17
I2.2.1 Program Model Phases and Characteristics	I-19
I2.3 Lessons Learned	I-20
I2.3.1 Lesson 1	I-20
I2.3.2 Lesson 2	I-21
I2.3.3 Lesson 3:	I-21
I2.3.4 Lesson 4	I-22
I2.3.5 Lesson 5	I-23
I2.3.6 Lesson 6	I-24
I2.3.7 Lesson 7	I-24
I2.3.8 Lesson 8	I-25
I2.4 Challenges	I-25
I2.5 Conclusion	I-25
I2.6 About the Author	I-26
I3.0 Tab 3: Electronic Combat Model Re-engineering	I-27
I3.1 Executive Summary	I-27
I3.2 Re-engineering Legacy Systems	I-27

I3.2.1	Maintainability Index and Metrics	I-28
I3.2.2	The Role of Software Architecture	I-28
I3.2.3	COTS (Commercial-Off-the-Shelf) Software and Ada.....	I-28
I3.2.4	Dual-Use Opportunities	I-29
I3.3	Summary	I-29
I3.3.1	Project Background	I-29
I3.4	Project Evolution — Translating from Fortran to C	I-30
I3.5	Research Study — The Future of IMOM	I-31
I3.5.1	Re-engineering IMOM.....	I-32
I3.6	Measures of Success — Speed of Development	I-33
I3.6.1	Measures of Success — The Maintainability Index	I-34
I3.6.2	Measures of Success — Software Complexity	I-35
I3.6.3	Measures of Success — Module Maintainability	I-37
I3.7	Software Reuse	I-37
I3.7.1	The Benefits of Ada	I-39
I3.7.2	Ada and COTS	I-40
I3.7.3	Dual-Use Potential	I-42
I3.8	Summary	I-42
I3.9	Bibliography	I-43

I1.0 Tab 1: The DoD Generic Fighter: F-22's Historical Foundation

Jon Floyd

Lockheed Fort Worth Company
F-22, LCSS IPT

Phil Gould

Lockheed Fort Worth Company
F-22 LCSS IPT Manager

Phil Mastrolia

SM-ALC, F-22 LCSS IPT

John White

F-22 SPO, F-22 LCSS IPT Manager

Presented at the Seventh Annual Software Technology Conference Salt Lake City, Utah, April 14, 1995

I1.1 Introduction

The “*generic fighter*” referred to throughout this paper is an invention of the F-22 Life Cycle Software Support (LCSS) Integrated Product Team (IPT). This generic fighter is an amalgamation or normalization of four modern front line fighter programs currently in service in the US Navy and the US Air Force — the F-14, F-15, F-16 and the F/A-18. The generic fighter was invented to allow reasonable comparison with the forecast software changes in the F-22 weapon system. The result of this comparison will allow an optimized support structure for F-22 software resulting in reduced life cycle costs. The data analyzed to create the generic fighter was gathered in response to 54 questions generated by the LCSS IPT during the first quarter of 1994.

I1.2 Generic Fighter Architecture

The generic fighter of the 70s and 80s utilized a federated avionics architecture (see Figure I-1). The Air Force fighters typically used MIL-STD-1750 processors while the Navy pursued the UYK series of standard processors. Bus systems evolved from one HOO-9 to one to five MIL-STD-1553 A or B bus systems. As time passed, more functions were automated and more analog functions were digitized resulting in a growing number of processors. By the late 80s, more functions were being merged in a single processing element as miniaturization occurred.

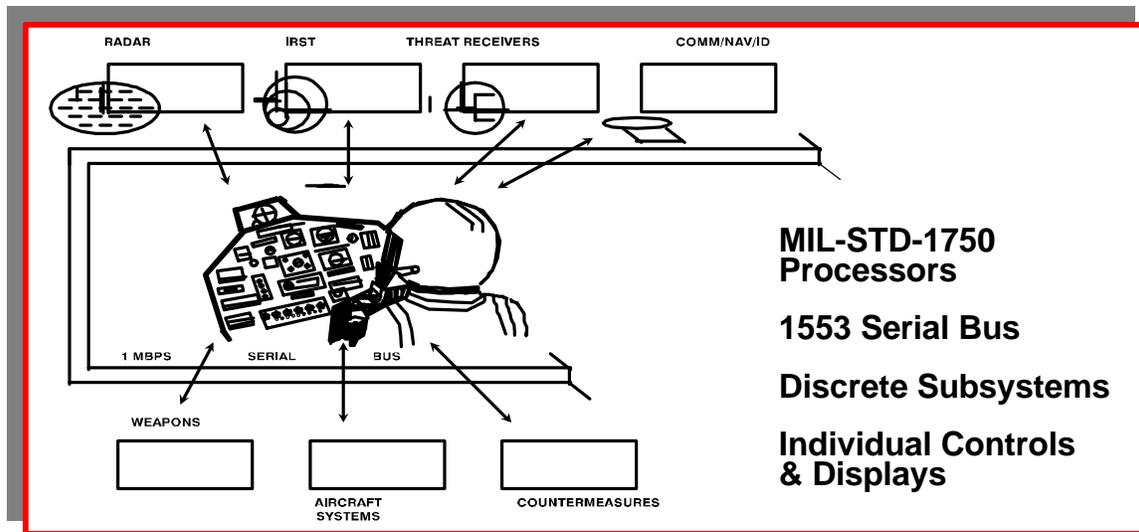


Figure I-1. Federated Systems

The generic fighter had four hardware unique baselines, with ever increasing computing resources. Each hardware baseline had multiple software variations. Even more variants existed if the aircraft was sold overseas. A single software baselines, compatible with all hardware configurations, was always the common goal but rarely achieved. The major impediment was shortage of retrofit funds caused by diminishing value of older aircraft retrofit.

I1.2.1 Variants

From the original hardware baseline there were new hardware baselines approximately every five years. New navigation, radar, electronic combat, and weapons were added at each new hardware baseline. Software baselines and releases followed a schedule that often was hardware driven (see Figure I-2). There was an average of three software releases the first year of service after PCA. The second year of service typically had two software releases. Eventually software releases settled to one every 18 to 24 months. As hardware sensitive software baselines matured, the release frequency decreased. Eventually software changes saturated the hardware capability and hardware upgrades were required. For each new hardware baseline, there were two to three corrective software updates released within the first 12 to 18 months.

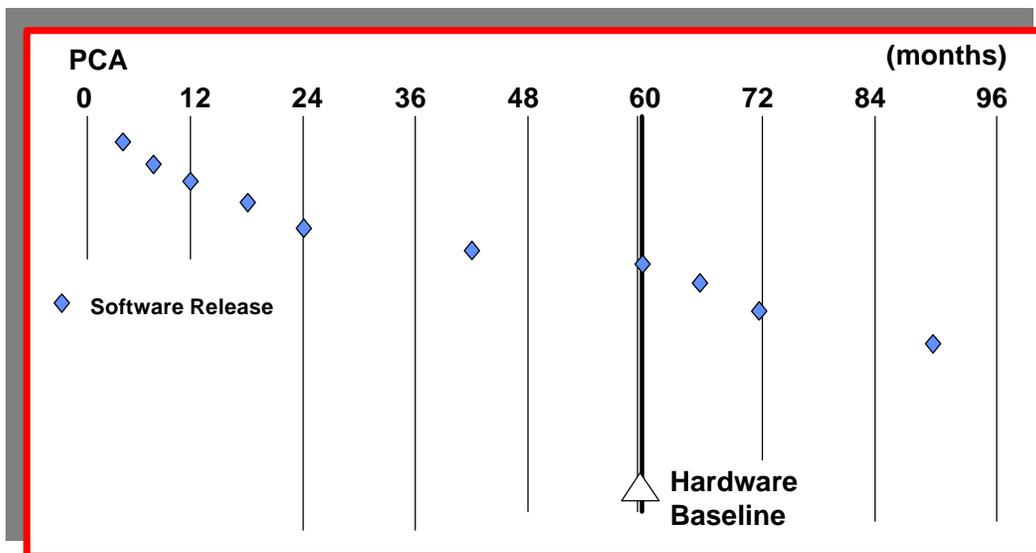


Figure I-2. Frequency of Software Releases

As the generic fighter matured, many variants of the hardware remained in the field. Each of these variants required a supporting software release. In addition, foreign military sales (FMS) typically required a software release of varying limited capabilities with each country requiring separate configuration management. This resulted in multiple versions of the software being maintained concurrently as the example shows in Figure I-3.

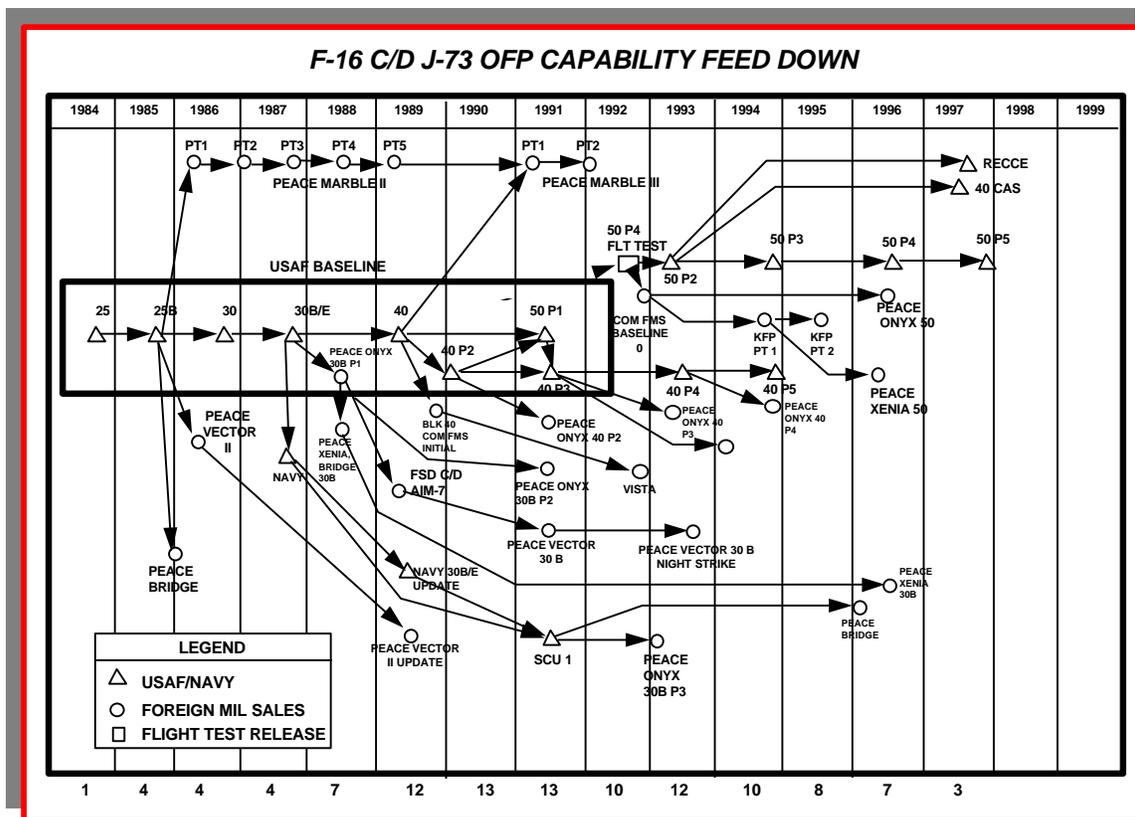


Figure I-3. Software Baseline Proliferation

The basic generic fighter started as an Air to Air fighter with less than 100K words of on-board memory. As more roles were assigned to the generic fighter, more systems were added and on-board memory was forced to expand. By the late 70s, the generic fighter had grown to approximately 300K words of on-board storage but with only 40 to 60% memory utilization. (These numbers are for avionics systems software only.) The mission related functions grew from 3 to 5% per year until the memory was forced to expand. By the end of the 80s, the on-board memory had grown to 2 million words with 70% used. (See Figure I-4.) Also in the 80s, non-avionics software began to appear. Digitized engine controllers, digital flight controls and diagnostic software were installed on the generic fighter.

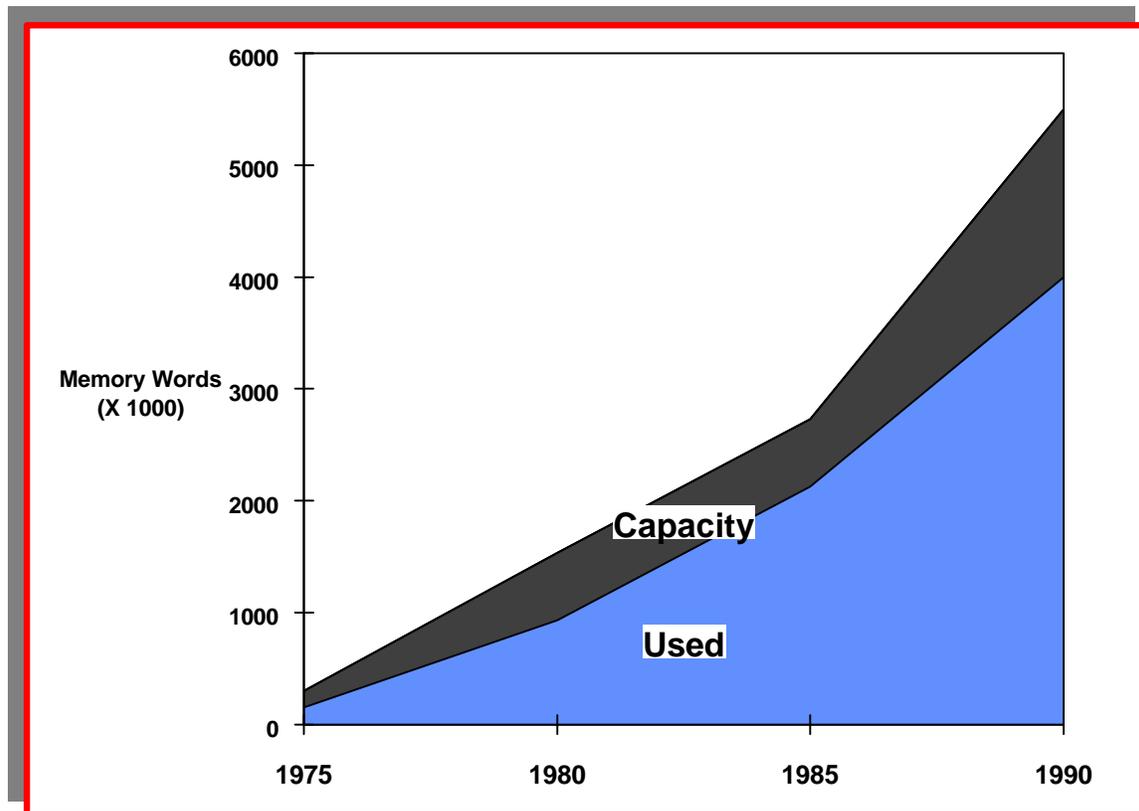


Figure I-4. Memory Growth Over Time

I1.3 Software Changes

After the first software baseline was established, typical avionics software maintenance changes can be categorized using the three basic types of software changes as defined by Swanson. Swanson's categories of software changes include corrective (fixing bugs, no new requirements), adaptive (adapting working software to hardware changes, no new requirements), and enhanceable (modifications/improvements of working software, new requirements). The percentage of change, by type, is illustrated in Figure I-5.

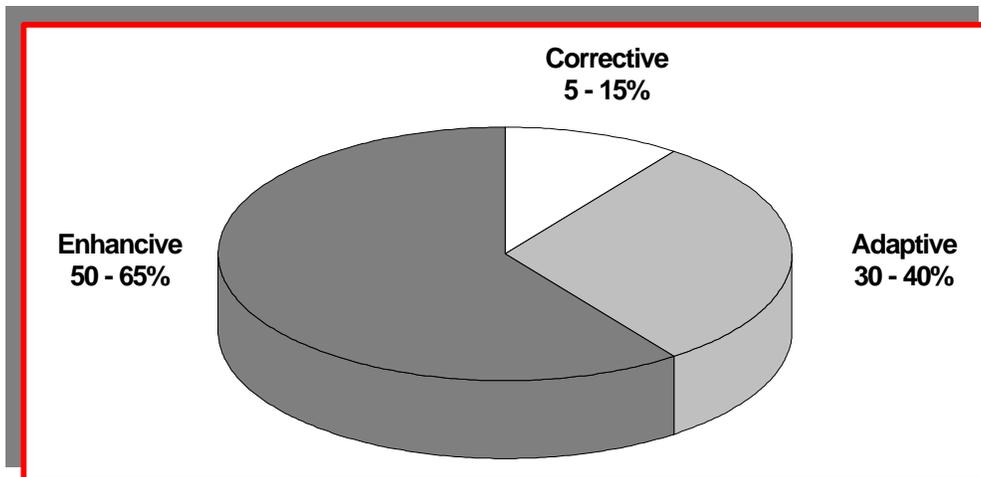


Figure I-5. Types of Changes

For a typical software block change, where no hardware was being changed, approximately 10% of the software was changed. This 10% was broken down as follows: 3-5% new code, 5-7% modified code. Approximately 88-93% of the software remained unchanged. The 10% rate of change's impact on the operational flight program (OFP) software configuration is reflected in Figure I-6. Software changes which were driven by hardware changes, either the addition of new hardware or the upgrade of existing hardware systems, had a much lower percentage of reusable software. Obviously, the amount of reusable code depended on the extent of the hardware modifications; expanding a memory board has a minor software impact compared to changing the processor type.

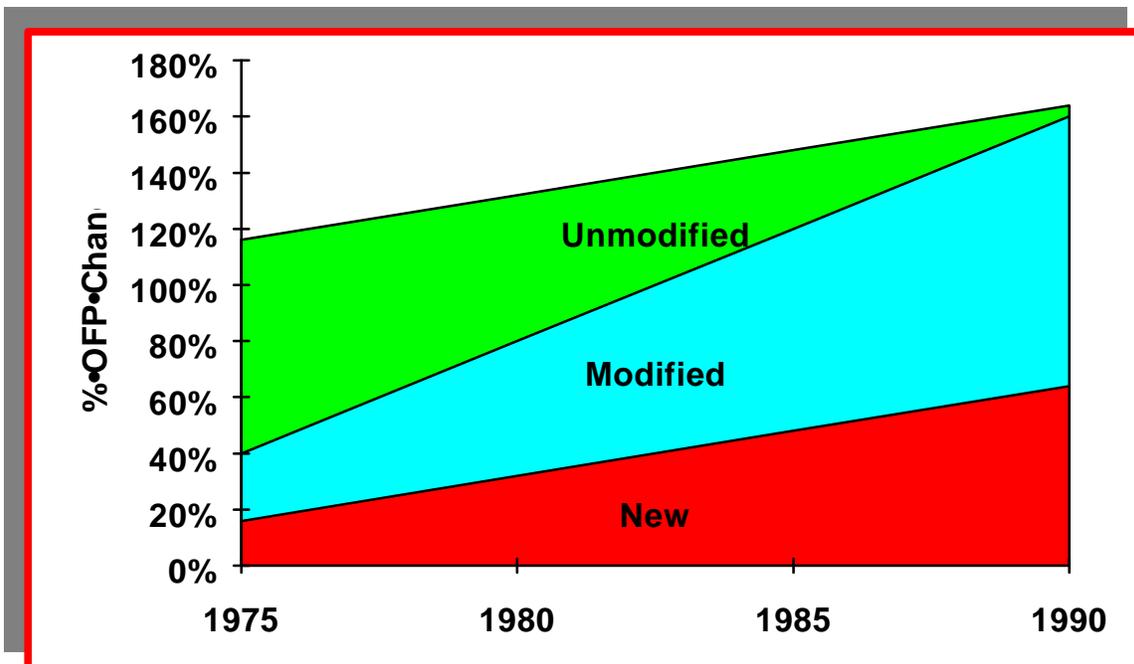


Figure I-6. Impact of Software Change on Operational Flight Program

Often the amount of software that was changed in a block cycle was limited by available funding or by available schedule. The larger and more complex software configuration items were typically most impacted by funding and/or schedule constraints. This is due to the increased integration and test requirements associated with large and complex software configuration items. Not all software changed at each block change. The generic fighter's primary areas of change were mission and human interface related (see Figure I-7). These primary areas of change included controls and displays (pilot/vehicle interface improvements accounted for up to 60% of the changes), stores addition, launch envelope improvements, radar improvements, electronic defense systems addition and improvements, and identification capabilities and improvements.

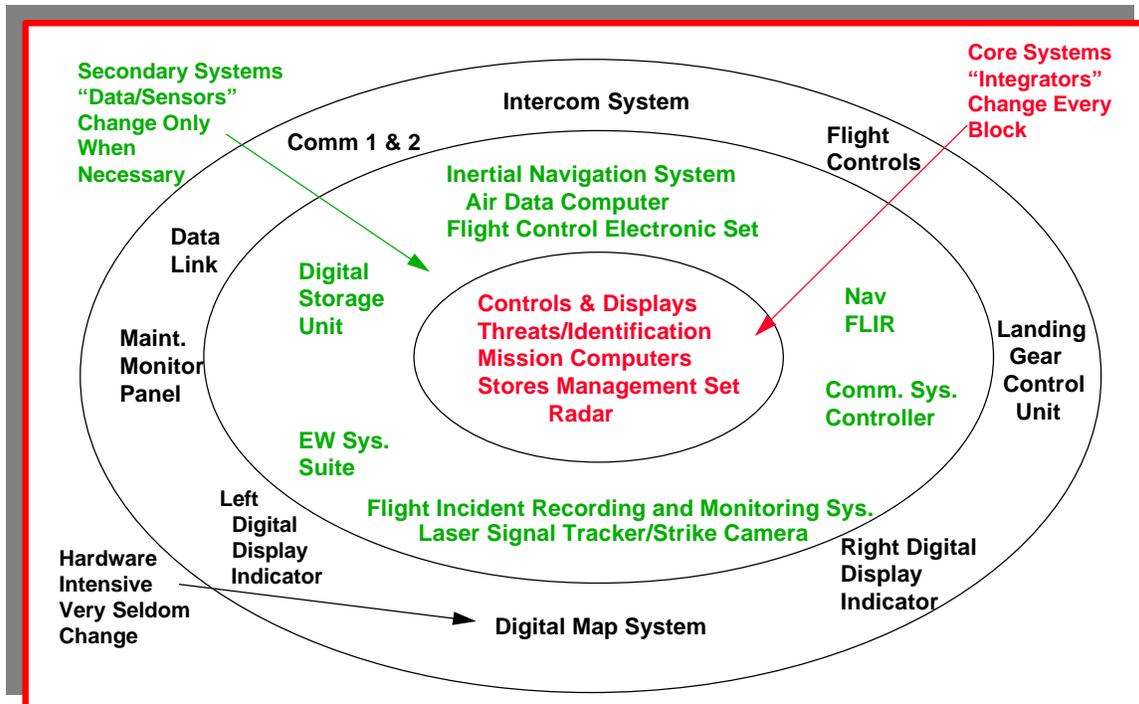


Figure I-7. Subsystems/Functions Affected by Software Releases

Some systems were very stable and exhibited little or no change to date. Examples of this were the inertial navigation system, air data computers, and digital data storage units. Software which was closely bound to hardware typically changed only when the hardware was upgraded. Examples of this were landing gear control, anti-skid systems-communications, and digital flight controls systems.

I1.4 Software Support Process

Software changes are categorized by the user as either routine (block change cycle), urgent (six to eight weeks), or emergency (within 72 hours). The goal of the generic fighter was to field a routine software release every 18 months; however, a single block change cycle required 36 to 44 months (see Figure I-8) from requirements definition to fielding. To accomplish this, as many as three or four software baselines were in work concurrently (see Figure I-9).

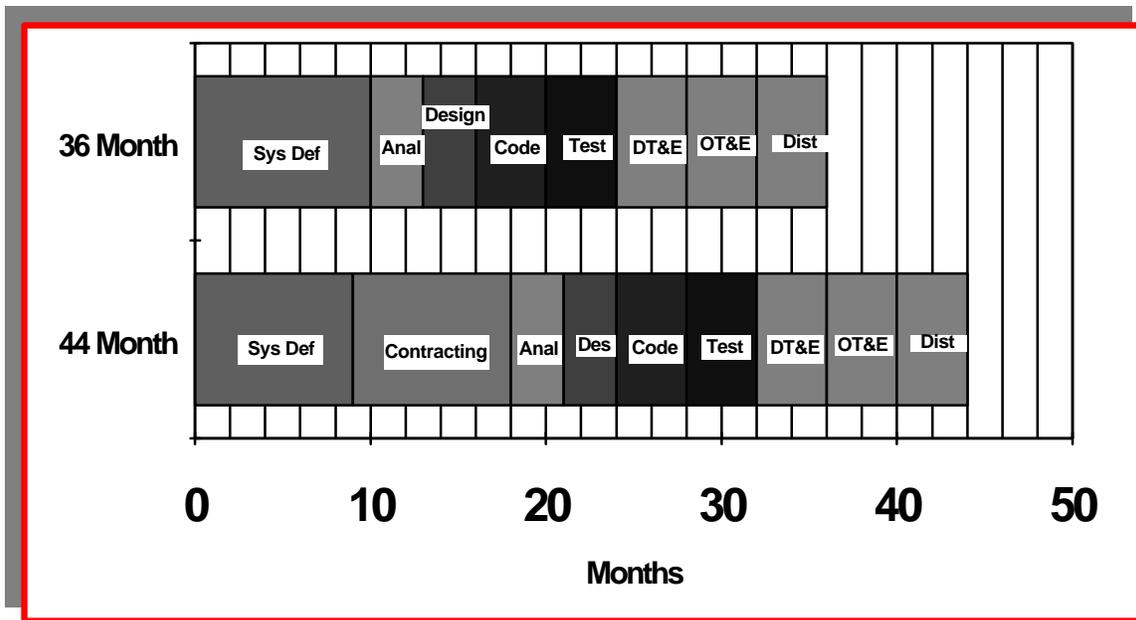


Figure I-8. Block Change Cycle

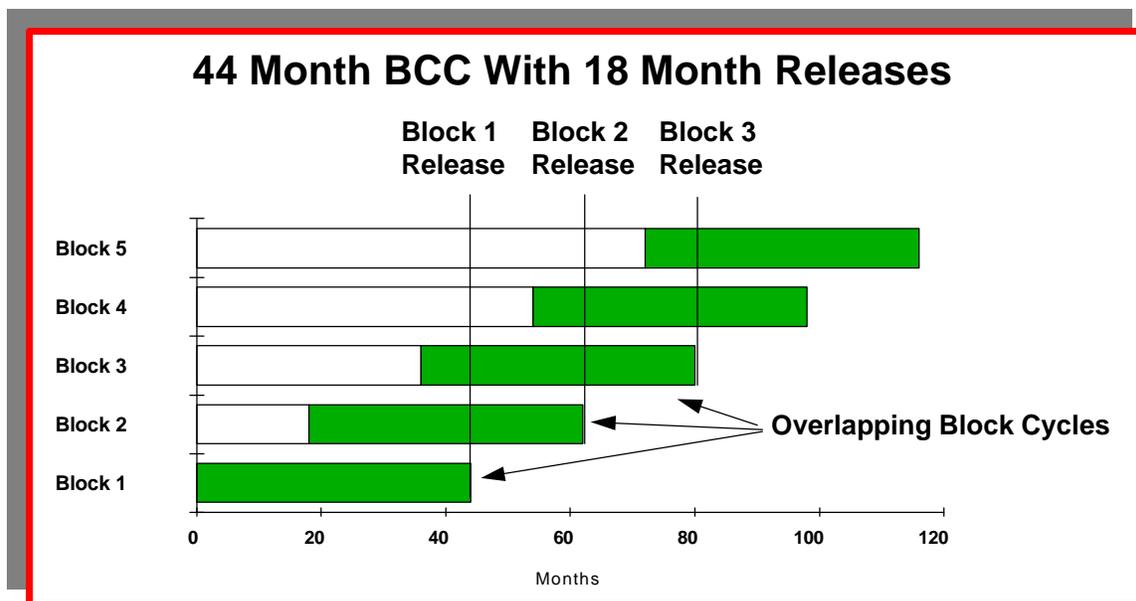


Figure I-9. Block Change Releases

The block cycle is divided into two phases. The first phase is the requirements analysis phase where the depot team, with the support of the user, develop an understanding of the types of system changes required for the next block change. This phase was typically undocumented and undisciplined. It required from 6 to 10 months for system definition/analysis and 9 months for establishing a contract with a software developer. The contracting timeline in the requirements definition phase was the primary driver in deciding whether the block change process was 36 or 44 months long. The second phase included allocating the changes to software configuration items, design, coding, integration, testing, and fielding the new release. This phase was in the process of maturing during the F-22 round robin with most programs actively pursuing a Software Engineering Institute CMM Level of 2.

I5.4.1 Support Costs

The generic fighter block change cycle required 400 to 500 man years and ranged from \$22M to \$200M with the normalized average of \$112M. This difference was caused by the length and size of the effort for the block change. All costs are included except for requirements elicitation and business development. The costs start with requirements identification through distribution of the software to the field. It also includes technical order (TO) and technical manual (TM) updates. The block change costs include systems engineering, software development, lab costs, flight test, TOs, and miscellaneous and are reflected in Figures I-10 and I-11.

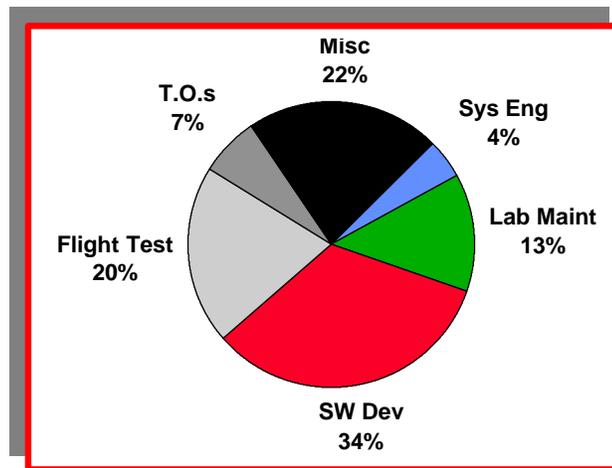


Figure I-10. Block Change Cost Allocation

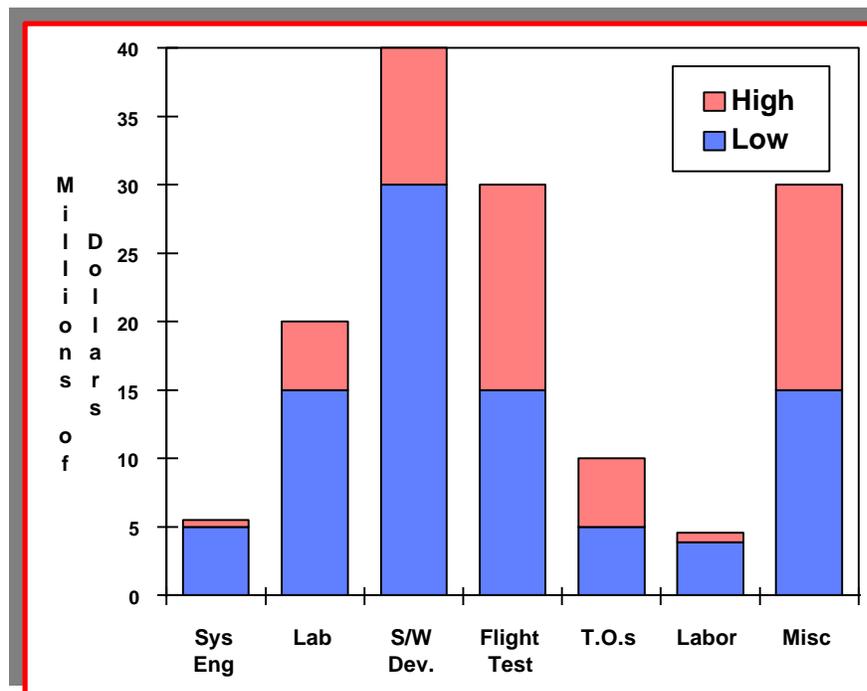


Figure I-11. Block Change Cost Ranges

The software development phase includes software requirements analysis, design, code, and unit test. For new hardware configurations, integration and system level tests took up to 40 percent of the total cost of development effort. For software only updates (without changing hardware) integration costs were less than 15 percent. Flight test costs were dependent upon location and duration. Typical flight test costs ran from \$30K to over \$50K per hour depending on chase plane and drone requirements. Other costs, included overhead, mechanical replacement, security, stores for test, and communications.

11.4.2 Support Strategy

The developing contractor has continued to provide critical skills at the government and contractor software support facilities. The production baseline of software was “owned” by the government at first production aircraft flyaway (FCA/PCA). Organic support for the first production baseline was not complete phased in until 5 to 8 years later. The initial program management directive (PMD) typically dictated a total organic effort for software support very early in the program. Over time it became apparent that this was technically impractical and undesirable from a program view. The contractor was continually adding capability, correcting errors, and optimizing the pilot vehicle interface. As a new avionics hardware baseline appeared, software support for the older baseline was transitioned to the government location.

Ratios of manpower mixes of 40%/60% to 60%/40% (organic/contractor) were common over the life of the program. The typical work split between organic and contractor resources by skill area is reflected in Figure I-12. This figure reflects the approximate mix at the ten year point of the generic fighter. Requirements analysis was done at the contractor’s facilities on simulators, reproduction and distribution is organic but was outsourced to subcontractors. Technical Orders/Technical Manuals were done at the Contractors facility.

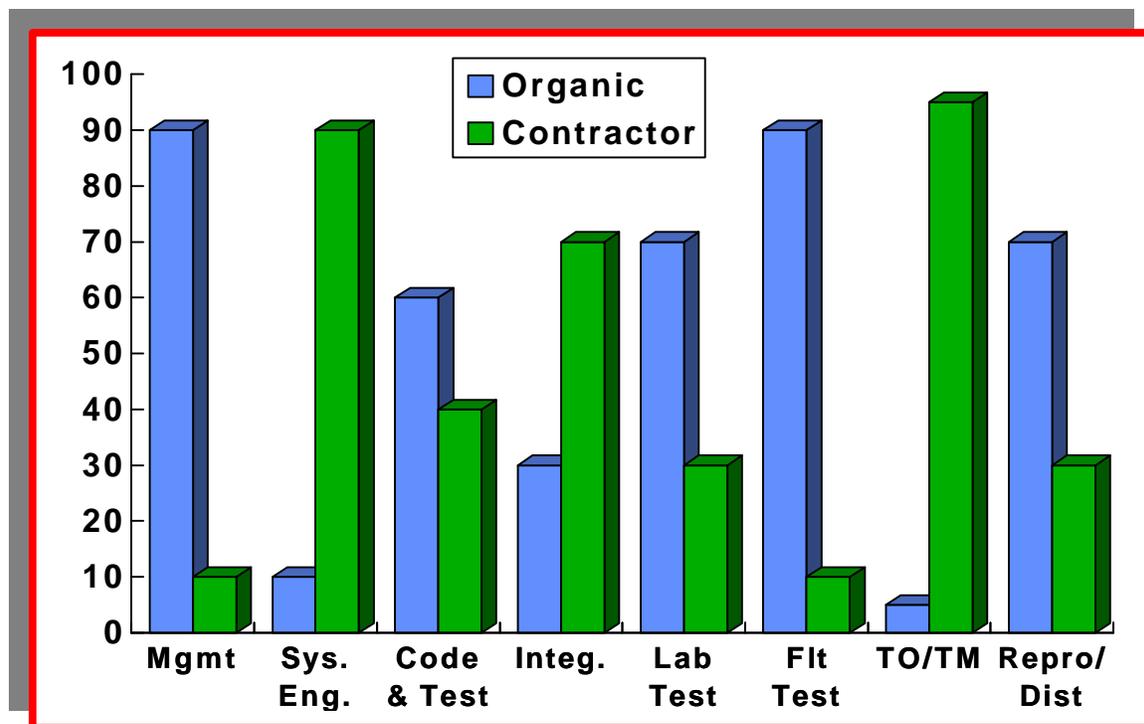


Figure I-12. Organic/Contractor Personnel Split (%)

Labor rate costs were as typically follows: \$110/hr for contractor personnel costs in a contractor-owned contractor-operated (COCO) facility; \$65/hr for contractor personnel costs in government-owned contractor-operated (GOCO) facility; and \$55 - 65/hr for organic personnel costs in a government-owned government-operated (GOGO) facility. The lower rates for labor performed in a government facility did not include total burdening.

I1.4.3 Support Environment

The generic fighter had few software development tools at the beginning of the 70s. In the OEM aerospace companies, an in-house software development tool making capability was developed. These “homegrown” tools were developed to fill in the blank spots between the available tools of the time. This was because few off the shelf software development tools existed during the early to mid 70s. In the labs little or no allowance was made for either growth or multiple configuration support. Non-reconfigurable lab equipment was a problem in the early period. Systems level testing was done without the aid of extensive simulation/stimulation and much of systems testing was done during extensive and costly flight tests.

By the end of the 80s (see Figure I-13), non-integrated computer-aided software engineering environments and multisystem labs with some simulation/stimulation capability were in use. “Homegrown” software development tools began to fall from favor because of the availability of COTS tools of higher quality, more functionality, and at lower costs. The short life span of these COTS development tools caused continual upgrade for development hardware and software. Simulations improved and began to replace trial and error, test and fix methods. This led to integration environments which were impacted less by each new hardware baseline, with very little impact to the support environment caused by software only changes. Occasional bottlenecks were encountered in the labs due to support of multiple support. Labs often cut holes in walls to test radar systems and hung weapons on the outside of the labs to simulate actual flight conditions. However, extensive flight testing was still required to ensure systems were meeting user requirements.

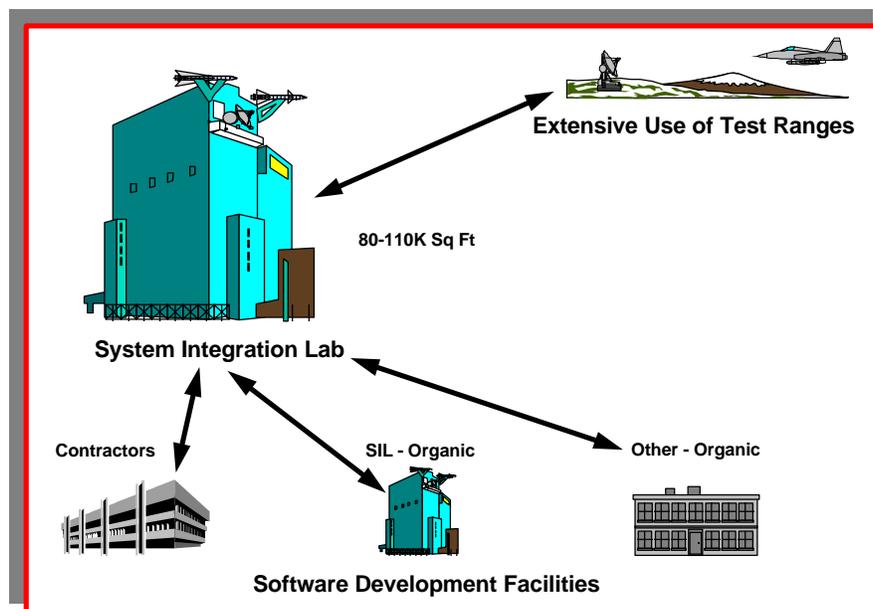


Figure I-13. Support System

I1.5 Training System Impacts

Training was separately contracted and, furthermore, was typically not integrated with the rest of Weapon System. Trainer and training systems typically lagged behind the fielded configuration. This was known by the user as “*version skew*.” The problem was that the acquisition strategy for trainers and training was different from, and at a lower priority than, the rest of the weapon system. Even if the trainers were at the same priority and received the OFP at the same time as the user, the training system required re-engineering before it could be used. Air vehicle software was not designed for trainer reuse. A lag of 2 years between new software installed in the fleet and trainer reconfiguration to match was not uncommon.

I1.6 Program Management

The generic fighter allowed patches but discouraged use of them. No patches were allowed for flight controls due to safety critical considerations. Flight Controls had the highest reliability but often took longer to develop. (Flight Controls typically had less than 1% error rate). Very few weapon system productivity figures were developed. However, the productivity numbers that were developed, varied from weapon system to weapon system and did not match IEEE productivity ranges for embedded applications. Mission related software changes had the highest productivity, safety critical software had the lowest.

Very little use of parametric models for project estimation. Developers tended to rely on past experience and rough orders of magnitude. Those projects using models such as COCOMO or LOCOMO tended to apply their own “*corrective factors*” to come up with numbers that matched their own experience. In many cases, very little data other than C/SCSC data was available which forced managers to make rough orders of magnitude estimations. Portions of the Block Change that tended to be fixed (e.g., lab overhead) were generally close to the budget. Portions of the Block Change that were product related (e.g., integration and test) tended to be several months late and over budget. If the project was schedule driven and a release was “*forced*” because of that schedule, two to three corrective updates were required and the total project cost increased by over 15%.

I1.7 Lessons Learned

These lessons learned were provided by the visited weapon systems during the round robin. Many of the weapon systems were experiencing the same types of problems and made the same suggestions. Below is a summary of the most repeated and critical suggestions provided to the round robin team.

1. Distributed support concepts (where maintenance is done on the same OFP in different locations) experienced major communications problems which led to misunderstandings of requirements and interfaces and thus to schedule delays and cost overruns.
2. “*Pure*” organic support of OFPs was technically impractical except for mature configurations no longer experiencing instability.
3. The maintenance planners must plan and provide resources for multiple concurrent software development efforts.
4. Technology insertion must be factored into the weapon system as early as possible.

5. Block changes will typically request maximum changes at the sacrifice of throughput and memory. Spare throughput and memory will typically be expended by the fifth block change (if not earlier). Plan for growth early or be caught in the trap of developing new capabilities at the cost of existing functionality.
6. Plan for changes in the support and training environments that are sympathetic to the OFP change. OFP changes can result in major changes to simulations and stimulations.
7. Training and technical orders must remain current with the OFP release.
8. Ease of system/software change depends on the quality of the documentation and when it was received. Development and design rationale is often more important than formal documentation.
9. There is a definite difference between the user and the supporter for the time requirements associated with a block change. The user expects to see an approved change fielded within 18 months. The supporter seems to feel that he meets the user's requirements by releasing a new version of software every 18 months. This often results in user priority requirements taking as long as 44 months to reach the field.
10. The amount of up-front time required to place a contractor on contract for a block change seemed excessive and often was the major cause for an eight to nine month slip in release. Contracting should be done in parallel with the requirements definition for the block update.
11. Every effort should be made to decrease the number of flight tests required for validation of a block change. Investments in comprehensive simulations and stimulations have a rapid return on investment by decreasing the number of flight tests required. However, these simulations and stimulations must also be updated as part of the block change process.
12. Labs that are given the same prioritization as operational aircraft have the capability to obtain necessary spares and replacement parts. Labs that are not given this prioritization suffered schedule slips and cost overruns due to a lack of operational equipment.
13. Those programs that co-located their integration and test facilities with their flight test capabilities significantly reduced flight test costs while improving schedule adherence and communication of requirements.

I.8 Acknowledgments

The F-22 Life Cycle Software Support Integrated Product Team thanks those who sacrificed their valuable time and participated in the F-22 Software Supportability Trade Survey: Col Ron Bischoff, SM-ALC/YFL; Capt Jeff McElroy, ASC/YFP; Capt Shawn Shanley, HQ USAF/AWC/28TS/TXBF; Capt Dennis Fleming, HQ ACC/SCTA; and Mr. Art Rindell, SM-ALC/TIS. The IPT would also like to thank project directors and those individuals residing with them at the SPO offices and system support facilities from each of the four aircraft weapon systems for accommodating our entire team and arranging their schedules to discuss the support details of their respective fighter aircraft: F-15 — Mr. Bob Anderson, WR-ALC/LFE; F-16 — Mr. Bruce Kress, ASC/YPVC; F-14 — Mr. Brad Gilmer, Code: P2205; and F/A-18 — Mr. Rich Bruckman, Code: C2107. Finally, a special thanks to the System Program Directors (SPDs) and Program Management Authorities (PMAs) for their support in this entire effort: Col. Rutley, WR-ALC/LF; Col. Kenne, ASC/YP; Capt Richard Evert, Program Executive Officer (PMA 241); and Capt. Joe Dyer, Commander (PMA 265).

I2.0 Tab 2: COTS Integration and Support Model

©Copyright Loral Federal Systems Manassas December 1994 [reprinted with permission]

Carolyn K. Waund

Loral Federal Systems-Manassas

I2.1 Abstract

Programs requiring high use of commercial-off-the-shelf (COTS) hardware and software are becoming more prevalent in the federal marketplace. Much of the emphasis on COTS solutions is due to increasing focus on information systems technology as we seek to re-engineer the government. Information systems technology is rich in COTS products and highly competitive, thus making powerful solutions feasible.

Loral Federal Systems (LFS) has in recent years moved to address this important systems integration market. Individual programs have achieved varying degrees of success in adapting traditional system development processes and management practices in the high-COTS environment. This program experience is a key resource. This paper highlights the results of an LFS initiative to use lessons-learned on recent COTS-based programs and defines a COTS integration and support model for guiding future programs. The initial model will be refined and matured by the LFS organization in the future. This paper provides: (1) observations about the current state of COTS integration, (2) a description of a model for a COTS integration and support process, and (3) a discussion of COTS program lessons-learned and their incorporation into the COTS process.

I2.1.1 Observations

Our observations about the current state of large-scale COTS integration and support are illustrated in the following graph. Note that the graph compares a “*traditional*” development program using few COTS products with three variations of high COTS content programs. We observe an important reduction in effort for high COTS, yet believe that more can be achieved. The “*dream*” program requires little effort to integrate and support a system. This may be possible today for a small, single computer system, but will not likely be achieved for the large distributed systems which serve an enterprise.

LFS has identified a number of COTS product characteristics which must be dealt with effectively to drive program effort to the “*achievable*” COTS level. The COTS product “*facts of life*” are that:

- They are not interoperable with other COTS,
- Their literature overstates their capability,
- They never exactly match users needs,
- Unique versions are costly,
- Upgrades are frequent and asynchronous,
- There is limited support for previous versions, and
- They are not Ada friendly.

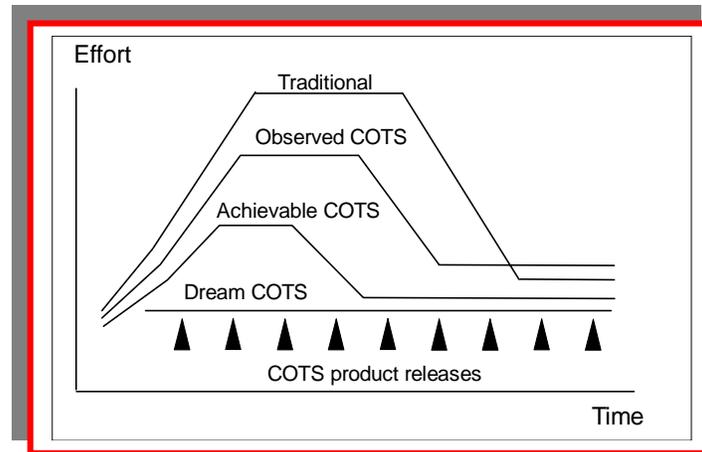


Figure I-14. COTS Integration and Support

The COTS product releases shown under the graph represent a primary characteristic of COTS products: they change frequently in response to the demands of the commercial marketplace. These changes begin to effect a COTS-based program when the products are first selected for system inclusion, and the effects continue throughout the system lifetime. The uniqueness of COTS-based programs is the inability to control the evolution of the commercial products which make up the system. Hardware product technology becomes outdated and the old products and parts reach end of life and are no longer available. Equivalent replacement products and parts may also not be available. Software products are regularly enhanced, correcting problems, and adding and repackaging functions. Support for back level versions is often not available; COTS customers are encouraged to incorporate the upgrades. This dynamic environment indicates the need for a continuing engineering analysis to refresh the COTS-based system.

I2.2 Program Model

The LFS approach has adapted a traditional, proven program model to one that supports a high content of COTS hardware and software. The model includes both technical processes and management practices. Actual program adaptations of the traditional model were analyzed. In some cases, the adaptations were successful and are retained in the current LFS model. In other cases the adaptations failed, necessitating a return to traditional wisdom.

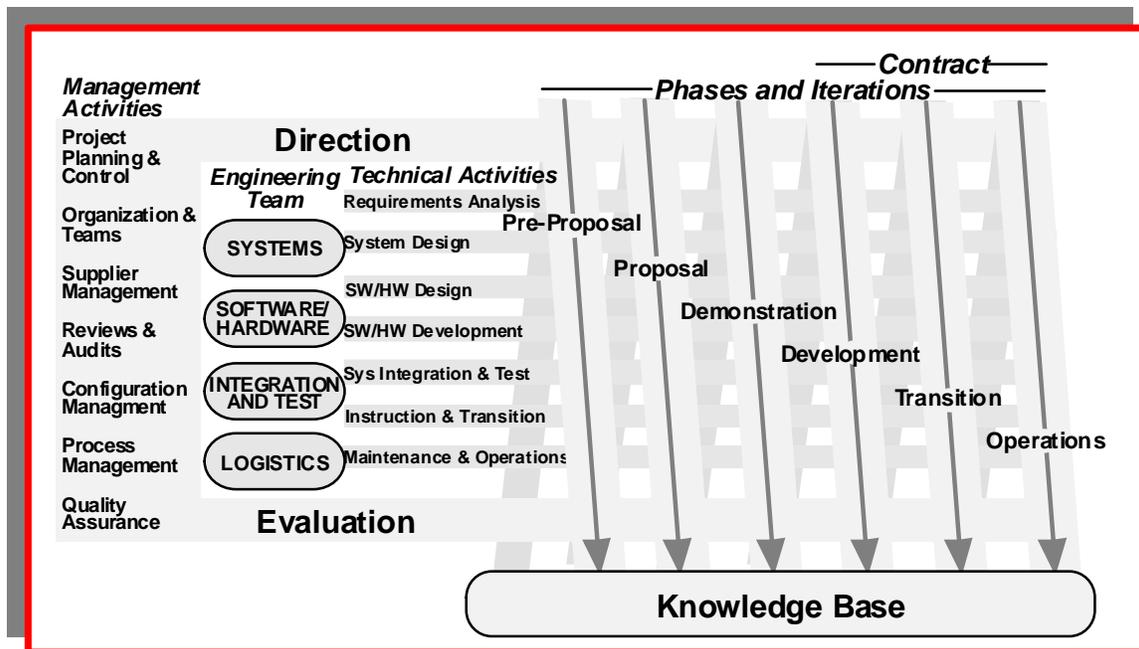


Figure I-15. Project Model

The model for COTS integration programs is illustrated in the following figure. It includes seven traditional technical processes performed in each iteration of activity. A concurrent engineering team, addressing all disciplines, is active throughout, coordinating all technical activities. Seven traditional management processes provide program direction, evaluation, and control. Iterations are grouped into program acquisition phases, beginning with pre-proposal and ending with operations. Contract start is at the beginning of the development phase. In each iteration, the emphasis of technical activities varies with the phase and objectives. A knowledge base captures information from all LFS division programs for use across the organization.

Each iteration begins by establishing objectives, based on the results and evaluations of prior iterations. Technical activities are scheduled with sub-objectives which, when completed, achieve the iteration objective. Each iteration ends with an evaluation of progress against iteration objectives and program goals. The evaluation includes actual and estimated costs and a revised risk assessment, along with recommended actions and objectives for subsequent iterations. These direction and evaluation activities are the key program control mechanisms. This program model is suitable for COTS-based system integration and support because it:

- Takes advantage of COTS product availability,
- Includes prototyping iterations to reduce risk,
- Recognizes that much engineering work is performed prior to contract award, and
- Accommodates COTS product upgrades asynchronous to the program schedule.

The table indicates characteristics of each phase and the integration objectives of iterations within each phase. The following paragraphs provide additional description.

For COTS programs, the pre-proposal iteration(s) determines the feasibility of satisfying program objectives and high level system and support requirements with products which can be commercially available in the needed time frame. This iteration emphasizes requirements analysis and system architecture tasks. Customers, systems integrators, and COTS vendors interact informally or via Requests for Comments (RFCs) or Requests

for Information (RFIs) during this time frame. The hardware and software product selection tasks are characterized by identifying that suitable options exist. Initial product selection is accomplished. Product experience or hands-on product evaluations are essential. Relying on product documentation or demonstration is an invitation to failure. The pre-proposal iteration produces the system integrator's initial system design. Results can be shared with the customer, for potential inclusion in the Request for Proposal (RFP).

The proposal phase iteration(s) adjusts the system design to RFP requirements and completes the hardware and software product selection. This iteration continues hands-on product evaluation and, if time permits, begins to integrate a prototype of the system. It is important to select the most challenging aspects of the implementation for prototyping priority. Early integration must focus on areas where the return in terms of risk reduction is the greatest. If integration problems arise, it is easier to accomplish a product change-out or architecture change in this time frame than in a later one.

I2.2.1 Program Model Phases and Characteristics

The demonstration phase can significantly reduce the risk in the requirements baseline. This iteration(s) firms up the hardware and software product selection. In demonstrations, customers can evaluate the human/computer interface, explore the process and data model, and assess consistency with the operations process. If time permits, prototyping can integrate the selected products in vertical and horizontal dimensions, creating a fairly complete system implementation.

Contract work begins in the development phase. Early formal reviews serve to baseline the system requirements and design, including the COTS product selection. This is an opportunity for the systems integrator to suggest requirements changes which will increase COTS content and reduce development content on the program, thereby reducing cost and risk. Prior to this review, the precontract customer evaluation prototype will be reviewed for requirements and/or design change. In this way, the results of prototyping activities are fed back into the requirements, which should remain flexible until the design validation is complete. System capability and functionality are developed during the remaining development phase iterations, which may, if necessary, incorporate COTS product updates. Formal testing culminates each iteration. On the final iteration, the system is ready for customer acceptance.

In the transition phase, the initial iteration produces a fully integrated and tested system configuration installed at a customer test or evaluation site. After the evaluation period, additional iterations produce system configurations at operational customer sites or platforms.

In the operations phase, iterations are driven by requirements changes, problem fixes, technology insertion, and COTS product upgrades. Depending on the significance of the changes, an iteration may be scheduled for prototyping prior to full implementation of the change. Note that COTS-product-induced perturbations are not exclusive to operations phase. They may occur in any of the preceding phases, and need to be handled when they occur.

The COTS model features early iterations through requirements, design, and prototyping tasks, encouraging requirements modification to achieve program goals and minimize risk. This iterative process assures that the final requirements are consistent with COTS content goals. If, for example, the goal is to build the system with currently available COTS products, it is important that the requirements reflect existing product capabilities. Appropriate adjustments to requirements can be made after COTS products have been identified, evaluated, integrated, and used in a prototype application.

PROJECT PHASE	CHARACTERISTICS	ITERATION OBJECTIVES
Pre-Proposal	<ul style="list-style-type: none"> • Customer/industry interaction • Request for Comments (RFC)/ Request for Information • Draft Request for Proposal (RFP) • Initial System Design 	<ul style="list-style-type: none"> • COTS feasibility • Hands-on product evaluation • Initial product selection • First integration • Initial operational analysis • Architecture definition
Proposal	<ul style="list-style-type: none"> • Formalized customer/industry interaction • Request for Proposal (RFP) • System design adjustment • Program planning 	<ul style="list-style-type: none"> • Hands-on product evaluation • Integration prototype • Big product finalization • Cost model developed
Demonstration	<ul style="list-style-type: none"> • Formalized customer/industry interaction • Live Test Demonstration (LTD) • Design validation • Best and Final Offer (BAFO) 	<ul style="list-style-type: none"> • Architecture validation • Integration prototype • Capability demonstration • Final bid product selection • Cost model refined
Development	<ul style="list-style-type: none"> • Contract starts • Extensive customer interaction • Complete implementation • Formal test 	<ul style="list-style-type: none"> • Extensive operational analysis • Business process reengineering • User interfaced prototype • Full integration • Phasing of capability • Acceptance
Transition	<ul style="list-style-type: none"> • Test and Evaluation • Installation • Replication 	<ul style="list-style-type: none"> • Evaluation site integration • Test site integration • Operational site integration • End user training
Operation	<ul style="list-style-type: none"> • Engineering Change Proposal (ECP) • COTS end-of-life and update (may occur in earlier phases) 	<ul style="list-style-type: none"> • Prototype of changes • Integration of changes

Table I-1. Program Model Phases and Characteristics

I2.3 Lessons Learned

To formulate the COTS program model, the following key lessons-learned on previous programs were addressed. For each lesson, we indicate how it is incorporated in the COTS program model and approach. In conclusion, we note barriers or challenges to incorporating these lessons, acknowledging that managing COTS programs is an exercise in tradeoffs; there is not always a single “best” answer!

I2.3.1 Lesson 1

Most shortcuts through the traditional systems development process have proven faulty, indicating a need to return to a disciplined, but tailored process.

The COTS program model is adapted from traditional, proven LFS technical processes and management practices. Activities for COTS have been adjusted, scaled down or up, but not eliminated. For example, the traditional software development activity was redefined as COTS software product installation and customization. This requires significantly less effort than developing the product functionality from scratch, so it is a “scaled down” activity. On the other hand, our approach features extensive integration prototyping, so integration is a “scaled up” activity.

The COTS program model features close coordination of technical activity among a multi-disciplined engineering team. Key team skills include systems engineering, software engineering, integration and test, and careful consideration of logistics concerns and the views of the end user, acquisition officials and vendors. This team effort facilitates communications and permits less formality in documentation and reviews. Team agreement, based on informal preliminary documentation of requirements and design, is generally sufficient to establish technical baselines. With rapid turnaround in system development (relative to traditional non-COTS development programs), rapid decision-making is critical. This multi-disciplined team is essential.

I2.3.2 Lesson 2

Some requirements should remain negotiable until COTS system design is validated via prototyping.

To reduce program cost and risk, requirements which drive unique non-COTS development and are not essential for the system's success should be candidates for change. The COTS program model uses iterations to formalize the feedback from system design, COTS product selection, and prototype integration activities to aid requirements analysis. This feedback can specifically identify the requirements which are not capable of being satisfied using currently available COTS products. These are the requirements which drive program-unique development of product enhancements, including “glue” code to fix interoperability problems or selection of a high risk product. Some of the COTS product noncompliance issues and interoperability problems are discernible without prototyping. However, some issues are uncovered only during hands-on product evaluation and integration prototyping.

The COTS program model features a number of iterations prior to the baselining of requirements in the development phase. In the pre-proposal phase, the integrator should provide feedback to the government customer, indicating those requirements which drive non-COTS content. In the proposal and demonstration phases, the ability to communicate with the customer is limited. During these phases, the integrator builds up a set of recommended requirements changes which would increase COTS content and reduce program cost and risk. These are shared with the customer at contract award, for consideration prior to formal requirements baselining.

I2.3.3 Lesson 3:

COTS product selection and system design validation should include, or carefully waive:

- Hands-on evaluation of each product,
- Testing of each product-to-product interface,
- Prototyping of developed application-to-product interfaces,
- Testing COTS product portability to new platforms, and
- Vendor and product considerations beyond functionality.

The COTS program model features many early iterations, giving opportunity for hands-on evaluation, product interface testing, and integration prototyping. The emphasis of these activities is directed where the most benefit in terms of risk reduction can be gained. Bypassing these activities is acceptable for well-known products or previously-demonstrated interfaces, where the risk is assumed to be low. However, there can be a tendency to assume compatibility and underestimate the integration effort required.

Experience has shown that it is difficult to understand a product by talking to vendor marketing personnel or reading literature. Hands-on evaluation by the integrator permits a vendor-independent assessment of how the product meets the program requirements, and helps to avoid conflicting interpretations of requirements and what it takes to satisfy them.

Although some products are designed to interoperate, there is no guarantee that product-to-product interfaces will operate properly for the types of data to be supported or the environment of the program. Testing can determine if interface problems exist. Resolution may involve changes by one or both vendors or integrating mechanisms created by the integrator. Early problem detection can influence design and product selection and result in lower program effort and cost.

The boundary between developed applications and COTS products in the operational system can be a source of integration problems. This has been the case on a number of LFS programs using Ada. Ada interfaces or bindings to products are not as prevalent, highly-functional, or mature as those for C. If the degree of compatibility with application development tools and COTS products is overestimated by the integrator, the development effort can be larger than expected. Prototyping is indicated for unproven interfaces.

Sometimes the best COTS software product for the functional requirements has never operated on the hardware and operating system platform of choice for the program. In these cases, the integrator and the vendor agree that a product port is the best option. There is always some risk in porting a software package, especially if the product has never been ported before.

Considering product characteristics other than functionality can also be important in the decision-making process. Such aspects include current quantities in use, reliability, product support, and past performance of the vendor.

I2.3.4 Lesson 4

A well-defined architecture can lessen the impact of COTS product upgrades. Integration facilitators can reduce risk, but also have disadvantages.

In the past, the tendency was to integrate COTS applications in an ad-hoc fashion, creating *glue* code as needed to permit applications to interface with each other and external interfaces. Each glob of glue is unique to the set of interfaces between a pair of applications, and may require frequent modification as business needs and technologies change. Because this type of system is expensive to build and support, other mechanisms are preferred.

The architecture for high COTS content systems needs to consider a number of approaches which can facilitate the integration of both the COTS products and developed components that comprise the system. These approaches include architecture definition, standards, and frameworks, which are discussed in the following paragraphs.

The major step in creating a top level architecture determines the degree of integration needed among system components and establishes an integration strategy, which may feature using a number of integration mechanisms.

When integrating COTS and developed system components, there are five dimensions of integration to consider:

- **Data Integration.** Data created by one component are transformed and transported to another component for its use, in the format and context it requires.
- **Control Integration.** An activity or product of one component will cause the activation of one or more other components.
- **Process Integration.** The enterprise goals are translated into processing and storage decisions for all the components that participate in that goal. These decisions can influence control integration and data handling. “Workflow” is an implementation of process integration.
- **Presentation Integration.** The user interfaces have the same look and feel.
- **Platform Integration.** The COTS products and developed software components are independent of the platforms and operating systems inherent to those platforms. A heterogeneous system can result.

The required degree of integration in each of these dimensions is determined to influence the evaluation criteria for selecting the integration mechanisms, COTS products, and defining an architecture to isolate and minimize change as COTS products change.

Standards-based COTS product interfaces facilitate integration, but do not guarantee compatibility. Mature standards are unambiguous but tend to be complex, while immature standards still allow implementation flexibility. This permits different interpretations by vendors which can cause interface problems.

Integration framework products, such as those being used in software engineering environments, provide tool integration services which show promise of facilitating COTS application product integration. Usage of these framework products in this domain is not yet proven, but the integration framework concept and services constructs can be effective tools in system design.

Selecting a product suite from a single vendor, vendor partnership, or coalition is a low-risk integration approach which minimizes problems within the suite. Unfortunately, there may be no flexibility to add a best-of-breed product from outside the suite due to the closed nature of the design.

Product characteristics such as published Application Programming Interfaces (APIs), which reveal most product functions, many documented user exits, and source code availability, can facilitate product adaptation by the integrator to the system environment. The use of these characteristics, however, implies software development.

In the COTS program model, iterations through system design, product selection, and prototyping activities permit the evaluation of a variety of strategies for COTS-based system architecture and design. Various integration facilitators and mechanisms can be explored.

I2.3.5 Lesson 5

Significant advantage can be gained by reusing previously integrated solutions.

There is value in reusing previously integrated COTS solutions to reduce program cost and risk. As a group, the LFS divisions have in place a number of experience-sharing mechanisms which facilitate this type of reuse. Although LFS programs address a variety of information systems application domains, most

share a need for integrated platform services (including networking, distributed computing support, and other middleware functions). Standards-based integrated solutions in this area are particularly good candidates for reuse.

In some cases, an integrator can string together contracts with similar needs and reuse integrated solutions or solution parts. To extend the set of ready-to-use solutions, the integrator can invest in COTS product integration initiatives.

The COTS program model includes a knowledge base of product and integration experience that is contributed to by each contract or investment program and is available for use by all programs.

I2.3.6 Lesson 6

Vendor contracts are unique, complex business/ technical/ legal agreements which must clarify all requirements and expectations.

LFS has found that COTS vendor contracts must go beyond standard commercial license and services agreements, and become similar to major development subcontractor relationships. This applies when standard COTS offerings do not meet program requirements and must be enhanced. As a result, COTS vendor relationships become more complex than that of simple commercial product or service offering procurements.

These unique relationships are also more difficult to negotiate and manage because COTS vendors do not typically deal in this manner. Their business and technical processes are tuned to produce and support a standard offering, not to provide specialized solutions for a particular customer's system. Another contributing factor to negotiation complexity is the fact that many commercial vendors are not experienced in dealing with the Government. Unfamiliar military contracting concepts and terminology, in particular, can contribute to major vendor misunderstandings.

For each COTS program, LFS uses an experienced supplier manager who reports to the program manager. The supplier manager leads a multi-disciplined team which defines and negotiates vendor contracts and manages vendor contract performance.

I2.3.7 Lesson 7

Plan and budget for frequent and uncontrollable COTS product end-of-life and update events during all phases of the program.

The uncontrolled nature of COTS products is often not recognized and planned for. When not properly planned, significant cost impacts can occur. It is likely that after system delivery, COTS software product upgrades will be released by vendors and hardware products and parts will become unavailable. We recommend planning for the engineering effort required to ensure system integrity.

The COTS program model features continuing iterations in which COTS product updates can be scheduled for integration. The model handles each COTS product end-of-life or upgrade event as an engineering

change proposal (ECP). Notification of a product end-of-life event by the vendor is followed by impact assessment, selection of a replacement product, and planning for its integration into the system.

Notification of product upgrade initiates impact assessment and integration planning.

I2.3.8 Lesson 8

Cost estimation methods for COTS integration programs must be improved.

Industry-proven cost estimation models exist for traditional development programs with a high content of software development. These models have been calibrated with the results of many programs throughout the years. This is not the case for programs with high COTS content, where estimation models do not exist.

To increase the predictability of COTS programs, LFS has standardized the process on COTS programs and developed a program cost estimation model tailored for COTS integration. LFS will use this cost model to estimate new programs. Metrics collected from previous programs are being used to calibrate the model.

I2.4 Challenges

Many of the actions which must be taken to remedy the problems experienced on COTS programs involve a shift of effort into the pre-contract timeframe. This is challenging to systems integrators because it stresses the Bid and Proposal (B&P) budget in a constrained timeframe when customer communications are limited.

For example, prior to submitting the offer to the customer, it is often necessary to validate a significant part of the COTS system design via prototyping to reduce risk. This effort is costly, yet without this validation, the schedule and cost risk of the program under contract is increased, and may be intolerable.

It is highly desirable to adjust requirements based on discoveries during system design, product selection, and prototyping, yet this is not accommodated by the procurement process. RFP requirements are mandatory and inflexible during the proposal and demonstration phases when system design, product selection, and prototyping are accomplished. Requirements changes which could increase COTS content and simplify integration are not allowable in the precontract timeframe, and may be difficult to incorporate after award on a fixed-price contract. This may add unnecessary cost and risk to the program.

A large part of the challenge on COTS programs comes from a lack of widespread understanding and experience with the unique aspects of COTS programs in government and industry. General agreement has not yet been achieved on the complexity of designing and supporting a system made up of uncontrollable, commercially-driven elements.

I2.5 Conclusion

Improved COTS program predictability will benefit both government and industry. This COTS program model is representative of many challenging COTS programs and establishes a framework for assembling processes tailored for COTS. LFS is seeking process maturity by continuing to refine the model and standardize effective processes for COTS integration and support.

I2.6 About the Author

Carolyn K. Waund is a Senior Programmer in the Federal Systems Integration Laboratory at Loral Federal Systems in Manassas, Virginia. She is currently responsible for defining methods and architectures which facilitate COTS product integration and software engineering. Prior to its acquisition by Loral, she worked for IBM Federal Systems Company in Houston, Texas for 29 years. She has been involved in a number of systems integration programs, both for federal and commercial customers. Most of her career has been focused on *real-time* support of manned spaceflight, performing both systems and software engineering tasks for NASA Johnson Space Center. She received her B.A. in Mathematics from the University of Texas at Austin.

I3.0 Tab 3: Electronic Combat Model Re-engineering

Idaho National Engineering Laboratory

March 1995

I3.1 Executive Summary

As modern software systems become increasingly complex and critical, executives, managers, and technical team leaders are faced with ever more difficult choices. Regardless of your position in a government organization or a commercial business operation, the insights contained in this monograph can be of benefit, and can help you achieve and keep a competitive advantage. Decisions regarding the disposition of existing legacy software systems can have an enormous effect on the operations — even survival — of government and commercial organizations. Many organizations are grappling with trade-offs of retiring older software systems and moving into more modern and efficient architectures, while trying to find ways to leverage some remaining capabilities of legacy systems to reduce cost and risk of software modernization.

The United States Air Force (USAF) is facing these difficult decisions. In a cooperative effort with the United States Department of Energy's (DOE) Idaho National Engineering Laboratory (INEL), the Air Force successfully addressed many of the hard issues facing commercial executives and managers today. The lessons learned by USAF/INEL provide valuable insights and models for action for both government and commercial decision makers. Although the project involved the re-engineering of a military software system, the applicability of the experience is appropriate across many commercial domains. The critical necessity for migrating from older legacy software systems to modern software architectures crosses boundaries of virtually every application domain in the internationally competitive software marketplace.

I3.2 Re-engineering Legacy Systems

The Air Force was faced with maintaining a legacy system which provided important capabilities to its users. While the users were satisfied with the system, the Air Force was finding it increasingly expensive and difficult to maintain software written for a proprietary, “*non-open*” hardware platform. As maintainability of the code decreased, support costs increased, and reliability of the system deteriorated.

The Air Force initially directed the INEL to re-host the application, and move it to an open systems architecture. Basically, this effort involved translating the existing code from Fortran into C. Although the resulting code functioned properly, and the application was moved to an open systems architecture, the code itself was even less maintainable than the original Fortran software. As more functionality was added to the original application, maintainability of the software continued to decrease. INEL was then asked to conduct a study of the application, to include an analysis, evaluation, and recommendations to the Air Force as to future directions for the program. The recommendations for the future were to ensure the continuing satisfaction of the Air Force user community. The study showed the system needed to be re-engineered to provide for future user needs, and INEL recommended a blend of technologies and methods, including a layered, object-oriented software architecture, implemented in the Ada programming language. The resulting re-engineered system produced a threefold improvement in maintainability of software.

Additionally, the re-engineered system has enabled new functionality to be added in a fraction of the time required with the original and re-hosted versions.

13.2.1 Maintainability Index and Metrics

The decision to re-engineer a legacy software system is difficult to justify without appropriate metrics. As part of the USAF/ INEL justification, specific metrics were collected and analyzed to provide indicators of system maintainability and complexity. These metrics, in combination with independently developed polynomial equations, have been used by some organizations to calculate a “maintainability index” which provides an indication of the maintainability of a software system. Based on independent studies and separate work performed by the University of Idaho and verified in the field by Hewlett-Packard, the maintainability index confirmed INEL’s recommendation to re-engineer the application. This experience with the maintainability index, and the subsequent verification of its applicability, offer powerful insights for decision makers who are contemplating re-engineering of legacy systems. As the USAF/INEL project has shown, a maintainability index can help in providing sound economic justifications for the re-engineering of legacy software.

13.2.2 The Role of Software Architecture

Many legacy software systems have been developed using a functional decomposition methodology (if any formal methodology was used at all). Early generations of software systems typically make extensive use of proprietary platform features, operating system calls, and language-specific constructs which severely hamper maintenance, reuse, and portability. These fundamental differences in the underlying software architecture contribute significantly to the lack of benefits to be derived from rehosting or translating efforts, as the Air Force learned on this project. INEL used an object-oriented layered software architecture to achieve many of the benefits of their re-engineering activity. By using layers for the application, interface, graphics, operating system, and hardware, the development team was able to deliver superb benefits to the Air Force. The layered architecture also enabled the use of a hybrid of technologies, language, and methods, in a well-engineered development effort.

13.2.3 COTS (Commercial-Off-the-Shelf) Software and Ada

For government software developers and decision makers, particularly those in the US Department of Defense (DoD), current policy requires the use of COTS software products wherever those commercial products can satisfy DoD needs. In those cases where there are no adequate COTS products, new software must be developed in Ada, unless a waiver is obtained to use another programming language. On the surface, the DoD policy on COTS and Ada seems to be fairly straightforward, but as the Air Force and INEL discovered, a superficial implementation of the policy can be extremely costly — in terms of support and funding. INEL’s experience in evaluating the “*real*” costs and facets of using COTS software can provide government decision makers with substantial savings in choosing between COTS and new Ada code. Decision makers and software managers are deluged with new software product offerings which claim to offer powerful capabilities and benefits. Filtering through the claims and discerning the real benefits, then comparing those benefits with the cost/ benefit of a “*from scratch*” application is a daunting task — regardless of your application domain.

13.2.4 Dual-Use Opportunities

As DoD and other government budgets continue to shrink, it is becoming more important for all government agencies to work together to exploit technologies and programs of common interest. It is also critical for government agencies to exploit technologies which have applicability in major commercial markets — what is known as “*dual-use*” technologies. The USAF/INEL effort is a superb example of interagency cooperation, with benefits accruing to the users, the agencies, the government, and the taxpayer. Much of the code is available as “*public domain*,” government-funded software, with excellent applicability in major commercial domains. The combination of object-oriented methods, COTS software, and Ada makes this project an ideal candidate for technology transfer to the private sector.

13.3 Summary

This monograph is designed to be a valuable information resource for decision-makers, software developers, and users. Most of the issues addressed are generic in nature, and span a broad cross-section of software domains. This document is part of an ongoing series of monographs which will investigate the major software challenges and solutions required for viable modern complex software systems.

13.3.1 Project Background

The Idaho National Engineering Laboratory (INEL) is a Department of Energy (DOE) National Laboratory, located in Idaho Falls, Idaho. INEL is engaged in a wide variety of projects, ranging from computer and software systems development, to environmental programs, to energy generation technologies, and national and international technology transfer projects. The organization has a track record for delivering complex software solutions for a broad range of applications, in both government and commercial environments. This project, the Electronic Combat System Integration (ECSI), was initiated by the INEL in support of the US Air Force Information Warfare Center (AFIWC) at Kelly Air Force Base, in San Antonio Texas. AFIWC’s mission includes detailed electronic combat modeling support for a variety of Air Force organizations. One of the models developed by AFIWC is the IMOM (Improved Many-On-Many) electronic combat model. IMOM is an electronic combat modeling system, which supports air operations combat mission planning. Basically, the system helps combat pilots plan their missions in an environment of hostile electronic combat systems. For example, pilots planning combat missions would be very interested in knowing the range at which a hostile radar system would detect their aircraft, so that they could avoid detection during the mission. Furthermore, using IMOM, pilots can run different scenarios showing the difference in detection ranges that would occur if they changed the altitude of the mission profile (i.e., flying in at 500 feet instead of 1,000 feet of altitude can make a huge difference in detection ranges).

IMOM has also been incorporated into the Air Force CTAPS (Contingency Theater Automated Planning System). CTAPS is a command and control system developed by a joint Air Force/INEL team for managing complex air/land battle operations anywhere in the world. Various models of different detection devices and technologies can be depicted in the IMOM system, showing the range and capabilities of a variety of radars. Also, the effect of height above the ground of the sensor, and the effect of electronic countermeasures (ECM) can be determined. In this modern era of increasingly sophisticated detection systems and anti-aircraft technologies, the success and survival of US military pilots are heavily dependent on an accurate depiction of the expected coverage of electronic combat systems.

The benefits that IMOM provides to pilots of modern military aircraft were proven during the Persian Gulf War, where the system was used extensively by US pilots, under the auspices of the Air Force Sentinel Byte program. Sentinel Byte disseminates and displays intelligence and key early warning data for use by Air Force combat mission planners and pilots. At the time of the Persian Gulf War, IMOM was a Sentinel Byte application, having been moved from the AFIWC and CTAPS programs into the Sentinel Byte environment. The fact that IMOM could be moved from one major functional environment (CTAPS) to another (Sentinel Byte) is significant. Basically, IMOM delivered critically important capabilities to Sentinel Byte, virtually immediately, without additional development time and resources being required. Substantial additional functionality was then added to the core IMOM application, enhancing the value of the system to Allied pilots in the Persian Gulf War. This ability to move critical capabilities from one major functional environment to another has tremendous applicability in non-Air Force organizations. Similar benefits can be achieved by large government and commercial operations that are able to share and leverage common operational requirements and solutions.

The IMOM system was originally an AFIWC in-house program used in response to tasking and requests from other Air Force components. Due to its excellent performance and capabilities, IMOM became very popular with Air Force users from other commands. As a result of this superb performance and popularity, IMOM was distributed to a broad range of Air Force users, including pilots and mission planning personnel. The graphical representation of the various IMOM family of models allows users to take advantage of the capabilities of the system in a familiar context and manner — just as they would if they were working with a manual system of maps, charts, and markers. The color graphical interface enhances the fundamental capabilities of IMOM, with corresponding benefits to its users.

I3.4 Project Evolution — Translating from Fortran to C

The original IMOM capability was developed in 1984. Written in Fortran, the original version was hosted on a proprietary VAX/VMS platform, and used Tektronix PLOT-10/STI graphics. The software was designed using a top-down functional decomposition approach, with a high degree of machine dependency in the code. The IMOM system was a success from the perspective of the users, and the system established a track record of successful usage. Due to the long-term successful track record, IMOM users requested numerous enhancements to the IMOM model(s). As electronic combat threats and technologies evolved, IMOM users required additional functionality in the software. As a result of user needs and advances in electronic combat technologies, the original IMOM expanded quite rapidly, with substantial additions to the initial software system.

In terms of current open systems architectures, the original IMOM was most definitely a “*closed*” system. As the Air Force and the Department of Defense migrated toward open systems technologies in the late 1980s, it became clear that the proprietary IMOM architecture needed to be modified. In 1989, the Air Force tasked the INEL with modifying the IMOM system to enable it to operate in a UNIX/Windows environment, using standard GKS graphics. To accomplish the required modifications, INEL translated the original IMOM Fortran code to C, and moved the software to a Digital Equipment Corporation (DEC) workstation. In keeping with the Air Force’s tasking, the original software architecture was retained, and the majority of the translation from Fortran to C was accomplished using an automated translation tool. The end result of this re-hosting task was an open systems implementation of IMOM which ran on UNIX, incorporated GKS (Graphical Kernel System) graphics, and used a “*point and click*” user interface. As was

the case with the Fortran version of IMOM, the new C implementation was well-received by users of the system. Some additional functional enhancements were made to the C version of IMOM through 1991.

The original Fortran IMOM evolved as a baseline for the additional capabilities. In 1990, INEL added the functional capabilities required by the Air Force's Sentinel Byte program, eventually evolving the Fortran IMOM baseline through version 5.0. In addition to the Fortran enhancements, the C version of the IMOM baseline was upgraded to include the Sentinel Byte requirements. The additional models are different from IMOM, and have their own community of users, as well as a separate Air Force office. Those models, the COMJAM (communications jamming), PASSIVE DETECTION, and RECCE (reconnaissance) models, were all originally implemented in Fortran. Unlike the original Fortran versions of IMOM, the other models were not translated into C. The Ada versions were a result of the re-engineering effort conducted by INEL.

Although the various IMOM implementations were quite successful from the users perspective, the different language and platform implementations suffered from configuration management (CM) problems. In addition to the CM challenges, the software was becoming more difficult to maintain and modify. In 1991, the Air Force realized that the existing implementations of IMOM would not be adequate to support user needs into the future. The cost of maintaining and modifying the software was becoming unacceptable, and enhancements were exacerbating the complexity of the system. As a result of these concerns, and in anticipation of expected user requirements in the future, the Air Force tasked INEL with conducting a research study to ascertain the future directions of IMOM.

I3.5 Research Study — The Future of IMOM

INEL's research study included four primary objectives:

1. Determine the objectives and future goals for IMOM.
2. Identify current industry and DoD standards which could apply to IMOM.
3. Analyze the current IMOM implementations from a software engineering perspective.
4. Offer recommendations to the Air Force as to how to achieve the IMOM objectives.

Among the objectives and future goals identified for IMOM was the need to provide software which would be more maintainable and modifiable than the existing Fortran and C code. More easily maintainable code would allow the Air Force to minimize support costs without sacrificing functionality and reliability. More easily modifiable software would enable the AFIWC to keep pace with changing user needs as well as new and emerging technologies. The ability to migrate the software to more powerful computer platforms was also a sound objective for IMOM. The current industry trends and standards which could apply to IMOM included a wide variety of technologies and tools. Clearly, the need for an open systems architecture was an ongoing requirement for the future. In addition, the use of modern software development methods, such as object-oriented techniques, offered substantial promise for long-term IMOM usage and support. Other DoD and industry standards, such as the Ada programming language, were included in the INEL research study.

As the study progressed, it became clear that there were two major factors which had a direct impact on the design and structure of the IMOM software: the software architecture; and the programming language used for the implementation. The deficiencies of the original software architecture, with its reliance on hardware-specific features and operating system calls, were perpetuated in the translated C version of the code. INEL concluded that a continuation of the original software architecture would effectively preclude any major

improvements in the quality and maintainability of the IMOM software. Since INEL's study was conducted from a software engineering perspective, the role of programming language selection in support of a well-engineered software implementation was included in their evaluation of IMOM. In this context, INEL compared Fortran and C with Ada. As INEL applied and evaluated various software engineering principles (i.e., abstraction, information hiding, encapsulation, modularity, etc.), it became clear that the Ada programming language offered superior support for a re-engineered version of IMOM.

Ada also offered benefits in the application of an object-oriented design for IMOM. As part of their evaluation of object-oriented technologies and techniques, INEL noted the importance of applying object-oriented methods in a disciplined software engineering context, as opposed to focusing on overrated and highly abused object-oriented programming features, such as inheritance and polymorphism. The evolution of the various versions of IMOM proved the importance of a sound, flexible design as a key factor in obtaining the benefits sought by the Air Force in the areas of maintainability and modifiability. From a design perspective, as well as a software engineering perspective, Ada was a superior choice for the future of the IMOM application. INEL's bottom-line recommendations to the Air Force were as follows:

- Re-engineer and redesign the system,
- Use object-oriented technologies, and
- Use the Ada programming language.

13.5.1 Re-engineering IMOM

INEL produced a Software Development Plan which set forth and documented the process by which IMOM would be re-engineered. The Plan emphasized the use of sound software standards, such as Ada, as well as the disciplined application of software engineering principles. The development team followed an iterative life cycle approach, to ensure flexibility and fast response to changing user requirements. The redesign of the software used object-oriented (OO) analysis and design techniques with an implementation in Ada. A hybrid OO methodology was used for the analysis and design, drawing from a variety of well-known OO methodologies offered by Rumbaugh, Booch, and Coad/Yourdon. Basically, the INEL development team used the best features of these various methodologies and combined and adapted them to fit the requirements of the IMOM software redesign. The result of the object-oriented redesign of IMOM was a reusable layered software architecture. The layered nature of the new software architecture enabled INEL to clearly define the interfaces between the layers, and implement the various pieces and subsystems of the application in a highly modular manner. The clear delineation between layers and between modules within layers was an explicit design goal to enable ease of maintenance and modifiability of the IMOM code.

One of the major benefits of the layered reusable software architecture was the mitigation of the risk of using a mix of software technologies and methods. While INEL had recommended the use of solid technologies, such as Ada, object-oriented design, UNIX, GKS, and Motif, those technologies and methods had typically not been combined together all in the same system. The underlying software architecture allowed the use of a hybrid solution consisting of a mix of language, architecture, methodology, and technology.

Because of INEL's focus on a well-engineered layered software architecture, they were able to apply a wide variety of powerful technologies and methods in a disciplined and cost-effective manner. The variety of techniques, methods, and technologies were applied in a controlled and well-engineered process to produce the layered software architecture. Due to the solid success of the IMOM re-engineering effort, the Air

Force directed the INEL to proceed with the re-engineering of the other electronic combat models. These other programs included Ada versions of the COMJAM, PASSIVE DETECTION, and RECCE electronic combat models. All of these models were successfully re-engineered during the 1991 - 1993 time period, using the same layered, object-oriented architecture and Ada.

I3.6 Measures of Success — Speed of Development

Once the re-engineering effort was completed, and all of the IMOM “family” of models had been implemented in well-engineered Ada code, an analysis was conducted to measure the success of the program. Although IMOM users were quite satisfied, the Air Force needed to ascertain whether the fundamental IMOM goals of improved maintainability and modifiability of the software had been achieved. Over time, software systems which have been developed using top-down, functional decomposition approaches have experienced significant deterioration in terms of maintainability. This is due, in part, to the effects of adding additional capabilities and functionality, which result in substantial increases in the complexity of legacy systems. In software systems which have not been well-engineered, new levels of complexity are introduced as defects are corrected, leading to further deterioration of code maintainability.

The introduction of new complexity, combined with the demand for new functionality far beyond the scope of the original design, has an extremely detrimental impact on the ease with which the code can be modified. The Air Force recognized that the limitations of the original IMOM software architecture precluded the implementation of well-engineered, modular upgrades to the software. Although IMOM was meeting the current needs of its user community, the Air Force was anticipating problems which would limit responsiveness to future user requirements.

The translation of the baseline IMOM system from Fortran to C required 54 manmonths of effort. While most of the translation was accomplished using an automated translation tool, the final C implementation required a significant amount of “*clean up*” by the development team. By comparison, the re-engineering effort of the baseline IMOM capability from Fortran and C to Ada required 72 manmonths of effort. Both phases of the project (translation and re-engineering) used a 4-person staff of developers. The re-engineering of the additional derivative models in Ada required a total of 20 manmonths for all three models. These models included the COMJAM, PASSIVE DETECTION, and RECCE models cited earlier.

At first glance, the Ada IMOM re-engineering effort appears to have required an additional 18 manmonths of effort to deliver the same basic capability as the Fortran and C IMOM implementations. This is definitely not the case. The re-engineering project involved the design of an entirely new software architecture, as well as the development of highly modular, reusable Ada code. Basically, the re-engineering effort was an investment in the future, with an expectation of leveraging off that investment to accommodate user needs in a more cost-effective and reliable manner.

As evidence of the value and validity of that investment, the real payoff for the Air Force and INEL began to accrue with the implementation of the Ada versions of the additional models. The actual time required to implement each of the models in Ada was on the order of 5-6 manmonths per model. By comparison, if each model was re-engineered from scratch, they each would have required a level of effort comparable to the baseline IMOM system — the order of 70+ manmonths per model. This is a clear validation of the payoffs to be achieved with a well-engineered software development effort, with a deliberate commitment to design or maximum reuse. Although no specific metrics were collected for re-engineering, the additional models in any language other than Ada, INEL believes that it would have required significantly more time to implement them using Fortran or C. The end result is that, without the application of sound software

engineering methods, including a layered, object-oriented software architecture, and Ada, each of the additional models would likely have required the same number of manmonths as the original re-engineered IMOM implementation.

From a user perspective, the reduction in manmonths for each new model has important ramifications: the “*time to market*” or fielding of these critical capabilities can make an enormous difference in terms of lives and equipment or combat pilots and military mission planners. This factor is critical for most organizations, in both military/government and commercial markets. Actual tracking of IMOM project files showed that an interim release of the re-engineered Ada implementation contained 20% fewer defects than the C baselines. Members of the development team attributed the use of an object-oriented design and Ada as major factors in the reduced number of defects. The reduced defect rate was achieved even with the shorter development time for the Ada implementations.

13.6.1 Measures of Success — The Maintainability Index

INEL’s research study indicated that the use of well-engineered software architectures, combined with object-oriented analysis and design, and an implementation written in Ada, would result in software that was more maintainable than code which was developed using top-down methods written in other languages. INEL set out to verify that the expected results and benefits had been obtained. One of the most impressive and critical effects of the Ada re-engineering effort was the impact on the measured maintainability of the code. The maintainability index of the IMOM Ada code was more than three times greater than the index for the equivalent functions written in Fortran or C. This is based on several software metrics which were collected and analyzed by the INEL development team, leading to the calculation of a “*maintainability index*” for the fielded software.

INEL conducted a static analysis of the source code of the various Fortran, C, and Ada implementations. Using PC-Metric, a source code analysis program, the development team calculated the value of two widely known software metrics: Halstead’s effort/module; and McCabe’s cyclomatic complexity/module. These metrics were just two factors which INEL used to assess the maintainability of the IMOM family of software. Using the values from the Halstead and McCabe metrics, INEL then applied a set of field-proven polynomial metrics to calculate maintainability indices for each of the IMOM baselines. The polynomial metrics were developed at the University of Idaho and have been validated in the field by Hewlett-Packard (HP). Hewlett-Packard has set a “*maintainability cutoff*” of 65 on the maintainability index. Based on HP’s experience with software in the field, a software package with a maintainability index of less than 65 is considered to be “*difficult to maintain.*” HP’s separate evaluation and independent use of the maintainability index verified that it was applicable for HP’s software systems. INEL applied the maintainability index to the IMOM family of software systems to provide a comparison and perspective as to the maintainability of the Ada, Fortran, and C versions of the IMOM baseline. The Fortran and C versions showed an accelerated decline on the maintainability index, while the Ada implementations stayed constant. The Fortran and C versions of the IMOM software were becoming “*more maintainable*” as additional functionality was added. In contrast, Ada versions of IMOM exceeded the HP cutoff value, and stayed virtually constant, even with additional functionality being added.

13.6.2 Measures of Success — Software Complexity

Tracking the trends indicated by the metrics, INEL documented a dramatic increase in the complexity of the Fortran-based models of IMOM. The trends in the Fortran models showed a nonlinear increase in software complexity as IMOM evolved from version 4.0 to 5.0, with a corresponding projection for continually worsening maintainability over the life of the application. For the C versions of the IMOM baseline, the complexity was roughly the same as the Fortran implementations. This rough equivalency is due in part to the fact that the C code was derived from the Fortran software, using the same basic software architecture, indicating a functional equivalency between the various language implementations of IMOM. For example, the Fortran version 4.0 is functionally equivalent to the C version 1.0, and the Ada version 1.0.

The average cyclomatic complexity (a measure of the number of paths through source code) for the Ada IMOM baseline was slightly larger than 3, while it was more than 9 for the Fortran implementation. For the C version, the complexity metric was greater than 13. The several-fold increase in complexity for the Fortran and C versions in comparison to the functionally equivalent Ada code has a direct effect on the maintainability of the respective language implementations. The maintainability of the Fortran and C code markedly decreased over time, while the maintainability of the Ada code stayed virtually constant. The consistency of the maintainability of the Ada code indicated a significant reduction in the complexity of each individual module. Each IMOM baseline module written in Ada was smaller, simpler, and easier to read and understand than the equivalent programs written in Fortran and C. The bottom-line benefit for personnel engaged in software maintenance and modifications was that less effort was required to understand the function and execution of each subprogram.

The Ada IMOM version 2.0 has many more functional capabilities than version 1.0. Contrary to the trends of the Fortran and C implementations, the increased functionality in the enhanced Ada code has not increased the complexity of the code. The cyclomatic complexity of the Ada IMOM version 2.0 had the same relative magnitude as the Ada version 1.0, indicating that the modifications and maintenance of the code had little impact on the complexity of the Ada software. The number of unique operators and operands in the Ada versions of IMOM greatly increased in comparison to the Fortran implementations. The number of unique operators were 1.8 times higher in the Ada IMOM version 1.0 than in the Fortran version 4.0, and there were 2.5 times as many unique operands in the same Ada version. The total number of operators and operands also nearly doubled.

One of the primary reasons for the increase in unique operators and operands in the Ada code is that the Ada software has many more local variables, less globally accessible data, and passes many more formal parameters between modules. These facets, in turn, are “*by-products*” of the object-oriented design, and the layered reusable software architecture, where well-defined interfaces between objects, layers, and components are required. Basically, these characteristics of the Ada code reflect the application of the proven software engineering principles of modularity and information hiding.

Even with the additional volume of code in the Ada version of the IMOM baseline, the Halstead-estimated effort, as a complexity indicator, was 53% less for the Ada code than it was for the equivalent Fortran implementation. This is in spite of the fact that the Ada version has 63% more executable statements than the Fortran version, and more than four times as many subroutines. The fact that Ada source code, unlike many languages, is not terse, cryptic, or obscure, contributed significantly to the expanded number of statements and subroutines. The size of the Ada modules was much smaller than the modules written in

Fortran and C. The Fortran IMOM baseline was comprised of about 334,000 lines-of-code, with the typical use of global data access and Fortran subroutines that is common with functionally decomposed software designs. By comparison, the Ada implementation was comprised of 213,000 lines-of-code, which means that the Ada modules, although more numerous, were also much smaller than the equivalent Fortran components. The reasons for the reduced lines-of-code in the overall systems include the following:

- A significant amount of code was shared between modules,
- More streamlined implementation of required functions, and
- Changes in the functionality of the models.

The smaller code modules in Ada offer additional benefits to the Air Force in terms of maintainability and modifiability. Smaller, easier-to-understand modules enable the use and exploitation of the reusable layered software architecture, with explicit support for the object-oriented design employed by the INEL development team. The Ada modules are easier to work with, test, and modify, thus facilitating the iterative software life cycle approach chosen by the INEL team. The discrete and well-defined interfaces between modules prevents major surprises when the software is integrated and tested — an area that is historically a major source of delays and problems and cost increases.

Code modules of low complexity and size also have an impact on the effectiveness of all support personnel. Additionally, the number of people required to provide support, as well as the skill and experience of the support staff, are directly affected by the low complexity and size of the IMOM baselines. With small and easy-to-understand modules, new maintenance personnel can be brought up to speed very quickly, with minimal impact on the quality and responsiveness of support. Furthermore, with easy-to-understand code packages, highly-skilled senior software engineers who are expensive and in short supply, can be much more productive and efficient. Less senior personnel can be used to conduct routine support, error fixes, and modifications to the code, freeing the senior staff to address more complex support requirements. Finally, fewer people are required to provide on-going maintenance and support, with a corresponding reduction in resource allocation and funding.

By comparison, the Fortran and C versions of IMOM, due to a significant degree to their much greater levels of complexity, are very challenging and difficult to maintain and modify. Safe and reliable maintenance and modification of the Fortran and C code requires the allocation of very experienced, very costly, and very scarce software engineering talent. Even senior and experienced personnel will require much more time to assimilate and understand the intricacies and complexity of the non-Ada implementations of IMOM. For both government/military and commercial organizations, the ability to allocate top software talent to areas with a higher return on investment (i.e., the design of new systems) can have a profound effect on the efficiency of operations. In military commands, where deployment of systems like IMOM to remote areas of the globe is common, the capability to maintain and modify critical software systems is often measured in terms of lives and equipment. Making software easier to maintain and modify with fewer people and with people who are less experienced provides an enormous return on investment.

From an operational perspective, the combination of the layered software architecture, object-oriented design, and Ada has shown conclusively the effect of software as a “*force multiplier*” for the US military. Without that working combination of technologies and methods, the benefits of the USAF/INEL re-engineering effort would have been substantially reduced. INEL also studied the three additional electronic combat models which were implemented (RECCF, PASSRVE DETECTION, and COMJAM), to compare results on those derivative models. The trends discovered in the IMOM baseline also showed up in all three of the other models, including the following:

- The Ada version contained many more modules than the Fortran implementation. This is, to a significant degree, the result of a completely different software architecture being used for the well-engineered Ada versions of the models.
- The Ada code modules were smaller in size than the Fortran versions. The smaller modules greatly facilitate the understanding and maintainability of the software. The modules are easier to comprehend, modify, and reuse.
- Each of the Ada modules is much simpler than the Fortran modules, as measured by the cyclomatic complexity values.

I3.6.3 Measures of Success — Module Maintainability

To provide a more detailed examination of the complexity and relative maintainability of the various systems, INEL conducted another analysis using the same polynomial model cited earlier. For this analysis, however, instead of evaluating an entire model (i.e., all of the IMOM baseline), INEL calculated the maintainability index for each subroutine of each software baseline. In other words, each subroutine in the Fortran, C, and Ada IMOM baselines was evaluated, and a maintainability index calculated. Once each subroutine in each of the various IMOM baselines was evaluated and a maintainability index calculated, the modules were then categorized according to their respective maintainability values. Modules with a maintainability index of less than 65 (deemed unmaintainable by Hewlett-Packard) were categorized as “*low*.” Modules with a maintainability index between 65 and 85 were ranked as “*medium*.” Modules with a maintainability index higher than 85 were rated as “*high*.” The Ada modules in the implementations of the various IMOM models were substantially more maintainable than the modules written in either Fortran or C. Generally, nearly two-thirds of all of the Ada modules rank in the “*high maintainability*” category. Conversely, around half of all of the modules written in Fortran and C are rated as “*low maintainability*.”

There was also a downward trend in maintainability as the Fortran and C versions were modified and additional functionality was added. In both the Fortran and C implementations, the number of modules which fell into the “*low maintainability*” category increased over time. Conversely, the number of Ada modules which were rated for high maintainability remained relatively constant over time and with additional functionality. The INEL development team attributed the benefits of the Ada versions of IMOM to the emphasis on sound software engineering, the use of the layered software architecture, the application of object-oriented design, and the stability, clarity, and software engineering support of Ada.

I3.7 Software Reuse

Software reuse does not enjoy a commonly accepted definition. Basically, “*reuse*” means many different things to different people, depending on their area of expertise, technical background, etc. The premise and promise of software reuse has been analogous to a technical “*holy grail*” for many years. The DoD and other government agencies, as well as major commercial enterprises, have pursued high degrees of software reuse as a means of lowering software costs, increasing productivity, improving reliability, and leveraging investments in software. Unfortunately, the promise of software reuse has not been widely attained, due to a variety of factors and impediments.

The INEL development team was able to amply demonstrate that software reuse is not a myth, and that significant levels of reuse can be obtained, with corresponding benefits to both developers and users. As a foundation for obtaining substantive reuse, the underlying software architecture and software design are

critical. The INEL team designed the Ada IMOM baseline with reuse in mind, with the expectation of reusing significant amounts of code as new models were added to the IMOM family. The layered software architecture was a key element in the attainment of high degrees of reuse in the re-engineered Ada implementations of the IMOM family of models. The INEL development team, knowing that additional models and functionality would be required beyond the initial IMOM baseline, planned their software architecture accordingly. They structured the layers to enable high levels of reuse within the domain-specific portions of the applications, an used layered “*application frameworks*” to leverage their investment in common interfaces.

For example, by planning for and designing the system for reuse, INEL was able to develop common user interfaces (man-machine interfaces, or MMI) which could be reused for all of the IMOM models. Because of the structured, object-oriented layers of their design, the development team was able to simply concentrate on the engineering algorithms of each new model or function (RECCE, PASSIVE DETECTION, and COMJAM), without concern for the underlying software architecture and MMI.

The layers also provide insight into the “*type*” of reuse attained. Basically, reuse can be separated into two general categories: domain-specific, and general purpose. Using the “*Application Layer*” as an example, there was significant reuse across that layer which was domain-specific to the IMOM models. INEL achieved high levels of reuse in both the domain-specific and general purpose layers of the ECSI effort. The middle layers were analogous to “*application generators*” and were reused throughout every layer. For example, the “*Interface Layer*” provided reusable code which could be applied throughout the vertical range of layers. These generic reuse capabilities can provide powerful advantages for users and development teams, for requirements such as mapping tools. By reusing the common constructs for maps (i.e., bearing, heading, coordinates, etc.) the INEL team has established a generic mapping tool with a general applicability for a wide variety of domains. By using this tool, a developer would be able to quickly produce a capability equivalent to a software program of 50,000 to 100,000 lines-of-code. The tool has the additional advantage of being comprised of code which has already been tested and fielded.

Generally, INEL realized a “*reuse rate*” 65% which means that 65% of the Ada code modules were reused in one or more applications. The savings which resulted were impressive: without that level of reuse, each of the three IMOM models (other than the baseline) would have required at least another 100,000 lines of new code. Stated another way, a less structured and disciplined approach would have necessitated a minimum of 300,000 lines of additional new code development. The impact on productivity and “*time to market*” is the most obvious benefit of the reuse rates achieved by the development team. There was also a substantial benefit which accrued due to the reduction in testing and integration, since the reused modules had already been through that process. The reduction in risk, due to the virtual elimination of the new errors that would have occurred with the development of new code, was also noteworthy. The bottom line for the reuse of Ada code in the various IMOM models can be summed up with this observation by a representative of the INEL development team:

Due to the insight and foresight of the Air Force, we were encouraged to apply sound software engineering rigor and discipline. We were allowed to apply the appropriate methods in the up-front engineering of the system, to ensure a payoff to users in the future. If we had not been able to design for reuse, and actually take advantage of reused Ada code on this project, each of the derivative models would have taken significantly more time to develop. The results achieved on ECSI are a validation of the benefits and return on investment h can be realized by applying a soundly engineered mix of technologies and methods.

If a poor process had been employed for ECSI, and sound software engineering had not been used, each of the three additional models (COMJAM, RECCE, and PASSIVE DETECTION) would have required 72

manmonths of development time, for a total of 216 manmonths. By taking advantage of Ada code reuse (in support of the engineering process used by INEL), all three of those models were implemented in a total of 20 manmonths — less than one-tenth of the time without reuse.

As most software engineers and managers know, reuse can be achieved with software designs, as well as actual code. One of the highest payoffs can be achieved by reusing designs, including object-oriented designs. INEL recognized during the analysis phase of their effort that each of the four IMOM models had a wide variety of components which could be shared among the models. Each of the applications contained common or similar user interfaces, modeling algorithms, graphics, and electronic components. During the design phase, the development team focused on determining the appropriate abstraction for each object class and its respective role within each IMOM model. The level of reuse for each object was directly linked to its role within the various models. Some object classes had the potential for reuse outside the electronic combat application domain, others could be shared by two or more of the IMOM models, and others were highly specialized and specific to a particular role within a single model. By evaluating the specificity of the various object classes, the INEL development team was able to make decisions as to when to design for broad reuse (i.e., outside the electronic combat domain) and when to limit the design to a specific IMOM model. The development team discovered that most object classes in the IMOM domain were conceptually the same although their characteristics and roles within each of the models were different. When those differences were relatively minor, an object class was designed which would meet the multiple requirements. Even with the broader requirements, this approach added little additional complexity to the object classes.

The results of this approach were impressive. Interim releases of the four IMOM models indicate the benefits to be derived — the four models in those releases had approximately 112 object classes, of which 73 were shared between two or more software systems. Each class had an average of 1,200 lines of commented source code. There were 11 algorithmic or interface libraries which are shared between the models. INEL discovered that reuse is not the general panacea which is often touted as a goal for software development. For certain types of problems, and within specific application domains, reuse can improve the quality of software and enhance the development process. The greatest reuse is possible when sharing objects within the same specific application domain by a single software development team. Beyond those parameters, object reuse is still quite difficult to achieve.

13.7.1 The Benefits of Ada

Throughout this monograph, the benefits of Ada in terms of its explicit support for software engineering discipline and object-oriented design have been cited as major factors in the success of the USAF/INEL project. The Ada language features provided excellent support for the implementation of the layered software architecture, as well as the attainment of significant levels of code reuse. Furthermore, Ada's strong typing, parameter checking in subroutine calls, and array constraints contributed to the benefits achieved in the ECSI program. In addition to these powerful attributes and benefits, the clarity of Ada source code and the benefits of that characteristic must be specifically cited. The names of variables used in the Ada IMOM baseline are more descriptive of the actual items or phenomena being represented than comparable representations in the Fortran version. The Fortran baseline used primarily six character identifiers for variables and subroutine names, a limitation which was carried over to the C-based translation. The descriptors in the Ada version allowed the "real world" to be more accurately represented in the actual source code. The C version of the IMOM baseline was just as terse as the Fortran version, due to the fact that the C code was translated from the Fortran and retained the same brevity for variable names. C has

earned a reputation for being more terse than other programming languages, so it can reasonably be expected that a new version of IMOM written in C would still include terse names for variables.

The additional clarity of the Ada source code pays substantial dividends in the areas of maintenance and modification of the software. Maintenance personnel are able to relate or “*tie*” Ada source code to design documents and change requests more effectively and efficiently than with comparable Fortran and C implementations. The clear and descriptive nature of variable names in Ada help significantly in reducing error rates and the introduction of new defects during code maintenance and modification operations. Ada’s package construct also supports the use of layered, object-oriented, flexible software architectures. The Ada package basically consists of two parts: the package specification and the package body. The specification part of the package enables a sound engineering implementation of the software requirement, through its support of encapsulation, information hiding, abstraction, etc. Interfaces between packages are specified and verified before the package body is written. The inputs, outputs, and assumptions for each package are clearly delineated and established. The package specification provides a great deal of information to maintenance personnel, akin to a high-level summary of the characteristics of the package. In addition to defining module interfaces, package specifications also offer essential insights to the humans who must be able to understand the inner workings of the software modules to be able to maintain and modify the code in a reliable, cost-effective manner. Neither Fortran nor C provide any comparable capabilities to the Ada package specification. The clarity of Ada source code (Ada code is written in English, as opposed to obscure symbols and other representations, and resembles a structured English outline), combined with the structured nature of its package specifications, has a profound effect on documentation and software reuse. By minimizing the complexity and size of Ada code modules, Ada software implementations become virtually “*self documenting*” due to the explicit representation of the function and interfaces of the respective modules. The extensive information included as part of the package specification enables developers to ascertain the potential for the reuse of existing modules in other applications, without having to delve into the source code of the package body itself. The improvements in efficiency, accuracy, and productivity are substantial.

The INEL development team discovered that the learning curve for Ada is far less than perceived. The team was able to assimilate fundamental Ada constructs within a minimal period of time. Furthermore, the team was able to use and apply engineering discipline and methods using specific features of the Ada language. As the INEL team noted, “*You do not have to use all of the features of Ada to make effective use of the language.*”

13.7.2 Ada and COTS

As mentioned in the Executive Summary policy requires the use of commercial-off-the-shelf (COTS) software whenever they will satisfy requirements. For those applications where COTS is inadequate to meet DoD needs, new software must be developed in Ada. The USAF/INEL re-engineering project had some valuable experiences in effectively applying that policy. Many of the “*COTS or Ada*” decisions for the IMOM models were obvious. For example, INEL did not want to write its own operating system, or its own version of UNIX or Motif/X, Similarly, existing software interfaces or drivers to peripheral devices did not warrant the development of new Ada code. The flexibility of the layered object-oriented software architecture enabled the development team to mix and match the appropriate technologies and tools without sacrificing maintainability and the ability to add new functionality. To meet the requirements of IMOM users, the INEL team had to develop two “*spin-off*” products: the Data Stream Analyzer and FormBuilder. The Data Stream Analyzer was basically a generic parser which translated data from one format to another. This is a

valuable and essential capability for IMOM users, since different sources provide data in different formats. The Data Stream Analyzer takes care of converting differing formats into one which can be accepted by the various IMOM models.

In the case of the FormBuilder, the Air Force and INEL both knew that there were several COTS products on the market which provided the capability which was needed. The basic requirement was for an application programmers interface for developing Motif-based graphical user interfaces (GUIs). At first glance, it appeared that simply buying a COTS form-building tool would be the best choice for the Air Force, and in keeping with DoD policy. Fortunately for the Air Force (and the taxpayer), INEL went beyond a “*first glance*” evaluation. The INEL team used GKS (Graphical Kernel System) for a significant length of time on the ECSI project. Although GKS is a standard, the implementations of the standard by commercial vendors are not consistent. By using commercial product implementations of GKS, INEL and the Air Force were dependent on product changes and support from vendors, who were subject to commercial market influences and different business goals. This risk was realized by the development team when the vendor of the GKS product being used for the ECSI project suddenly decided to discontinue marketing and supporting their implementation. INEL also did not want the Air Force to be subjected to the additional cost of GKS runtime licenses, so the development team built their own graphics package, called FBGraphics. For the Motif-specific requirement, INEL discovered that many COTS form-building tools worked effectively, and provided the capability to develop GUIs for applications. However, the tools all generated substantial amounts of code which was Motif-specific. This Motif-specific code, while performing the GUI tasks required for an application, was not itself readable or maintainable. INEL was thus able to justify the development of a FormBuilder written in Ada, which did not have the overhead of licensing fees and the generation of unmaintainable code. The development team required only two months to develop the Ada-based capability.

While the policy of “*COTS first, Ada second*” makes sense for many applications, there are significant hidden costs and other factors which need to be considered. For example, configuration management (CM) of an application can become a nightmare for program managers and maintenance personnel, if multiple COTS products are incorporated into a system like IMOM. As COTS vendors make changes and offer new releases, the impact of those changes can have serious and unforeseen effects on the rest of the system. If a fielded system uses several COTS products, the CM problems can be rapidly compounded over time. When evaluating a choice between COTS capabilities and new Ada software, decision makers must take into account much more than just the perceived “*up-front*” costs. In the case of the IMOM effort, INEL only required two months to produce a capability which could have been superficially satisfied by a COTS product, but the COTS solution would have been a serious detriment to the maintainability and modifiability of the IMOM models over time.

The IMOM family of models also required a database management system (DBMS) capability to meet user needs. In keeping with DoD’s “*COTS first*” policy, it has become popular and common for program managers to pursue the use of commercial database management products, especially relational DBMS capabilities. For the ECSI effort, the INEL team evaluated the efficacy of using a COTS DBMS versus writing the appropriate capability from scratch. Their solution for DBMS for ECSI was straightforward and easy to maintain, and consisted of ASCII flat files. This solution met user needs, and saved the Air Force substantial time and development resources.

I3.7.3 Dual-Use Potential

Since the software developed for the IMOM models has been paid for by the US Government, some of it is “*in the public domain*” and available for use by other government agencies and private enterprises. Some of the code has obvious commercial applicability and reuse potential, while other pieces of the IMOM applications are limited to specific domains. Clearly, the Data Stream Analyzer and FormBuilder tools are ideally suited for commercialization and exploitation by private companies. Similarly, other government organizations can derive substantial benefits from the use of these “*free*” tools. The user interfaces and graphical capabilities of the IMOM implementations have widespread generic applicability. The point and click nature of the user interface, along with the color capabilities of the various representations, are common features in most modern commercial systems. As an example of possible dual-use applicability, the generic mapping “*application generator*” cited earlier could be used by some commercial developers. Commercial developers who require basic mapping functions, as well as state and local governments engaged in economic development programs, are examples of organizations who could benefit from mapping software which is well-engineered and implemented in an international standard programming language. Other industries, such as aviation, marine, oil exploration, and land management/industrial planning operations could also make use of this generic capability.

By virtue of the partnership between the Air Force and the Department of Energy’s INEL, the process of dual use has already begun. The INEL has, as part of its mission, the transfer of technology to the commercial sector. Many significant software capabilities embodied in the IMOM models offer valuable features and benefits for commercial exploitation and use.

I3.8 Summary

The IMOM re-engineering project provides valuable lessons for both government and commercial software developers and decision makers. As modern systems continue to grow in size and complexity, the critical nature of well-engineered software in the success and survival of virtually all organizations is becoming more pronounced. The intelligent selection and application of solid methods and technologies are essential facets for progress. Key decisions related to the migration from legacy systems to more powerful distributed client/server architecture require sound justifications. The cost of maintaining and modifying legacy software is a critical factor in justifying a re-engineering effort. The “*hidden*” costs of providing adequate capabilities to the user base are often substantially higher than the explicitly measured costs of support.

The IMOM re-engineering project provided conclusive lessons as to the importance of software architecture in achieving significant software-related benefits. The program also showed that the application of object-oriented methods, in the context of a disciplined software engineering process, can deliver major gains in reuse, productivity, and maintainability. Finally, the IMOM re-engineering effort showed the value of Ada in obtaining maximum payoffs from an investment in software engineering practices. The benefits achieved in applying sound software engineering principles were multiplied by the explicit support that Ada provides for that disciplined approach. Simply stated, the magnitude of the benefits obtained would have been substantially lower if another programming language had been used. The IMOM formula for success was: Good people, with knowledgeable managers applying sound software engineering methods, implementing in Ada, with the needs of the user as the objective. That formula works for all segments of the global software community.

I3.9 Bibliography

- Coleman, D., "Assessing Maintainability," 1992 Software Engineering Productivity Conference Proceedings, Hewlett-Packard, 1992, pp. 525-532
- Idaho National Engineering Laboratory, "Improved Many-On-Many (IMOM) Model Research Study," EGG-EE-9555, Rev. 0, May 1991
- Oman, P. and J. Hagemester, "Construction and Validation of Polynomials for Predicting Software Maintainability," Software Engineering Tet Lab, Report #92-06 TR, University of Idaho, July 1992
- Welker, K., M. Snyder, and J. Goetsch, "Ada Electronic Combat Modeling Experience Report," Presented at OOPSLA '93
- Welker, K., "Application of Software Metrics to Object-Based, Re-engineered Code Implemented in Ada," Masters Thesis, University of Idaho, April 1994
- Welker, K. and M. Snyder, "Electronic Combat Model Re-engineering," ECSI Project briefing slides, 1994