

9-2009

Lessons Learned from a Large, Multi-Segment, Software-Intensive System

John T. Foreman

Carnegie Mellon University, jtf@sei.cmu.edu

Mary Ann Lapham

Carnegie Mellon University, mlapham@sei.cmu.edu

Follow this and additional works at: <http://repository.cmu.edu/sei>

This Technical Report is brought to you for free and open access by Research Showcase @ CMU. It has been accepted for inclusion in Software Engineering Institute by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Lessons Learned from a Large, Multi-Segment, Software-Intensive System

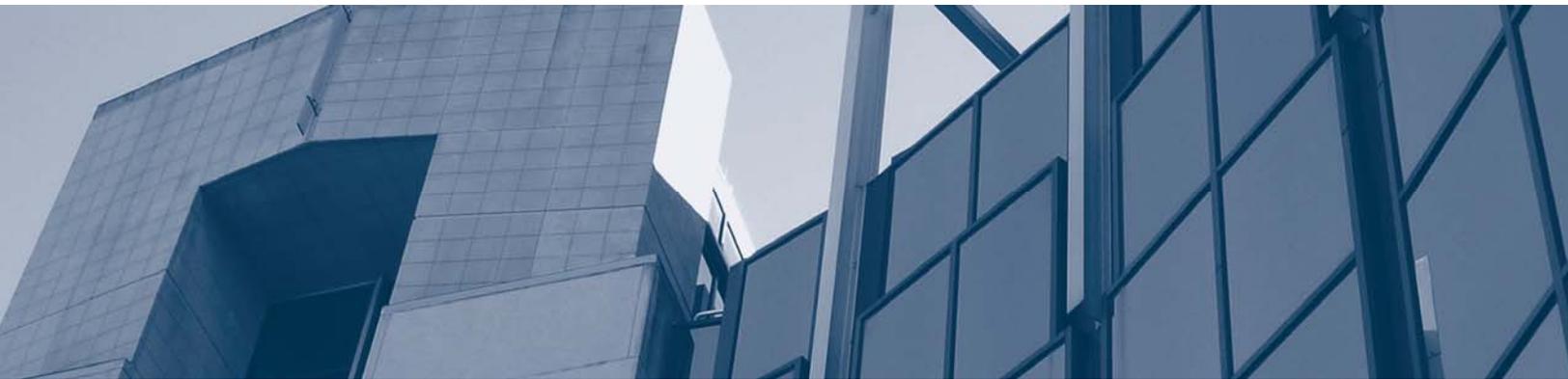
Mary Ann Lapham
John Foreman

September 2009

TECHNICAL NOTE
CMU/SEI-2009-TN-013

Acquisition Support Program
Unlimited distribution subject to the copyright.

<http://www.sei.cmu.edu>



This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2010 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (<http://www.sei.cmu.edu/library>).

Table of Contents

Acknowledgments	iii
Abstract	v
1 Introduction	1
1.1 Organization of the Paper	1
1.2 Observations Content	2
2 Observations and Associated Lessons Learned	3
2.1 Acquisition Life Cycle	4
2.1.1 Increase Number of Interim Design Reviews (IDR) and System Capability Reviews (SCR)	4
2.1.2 Capability Based Testing is “Test as You Fly ...” Testing	5
2.1.3 Inadequate Configuration Management	6
2.1.4 Acquisition Approach	7
2.1.5 Disregard for System-Level Software Considerations	7
2.1.6 Software Proposal Evaluation	8
2.2 Architecture	9
2.2.1 Administrative Decomposition of the System has Sweeping Technical Effects	9
2.2.2 Inadequate Software Architecture Focus	10
2.3 Requirements	11
2.3.1 Frequent Requirement Change Caused Significant Rework and Analysis Churn	11
2.3.2 Error Conditions Not Completely Defined	13
2.3.3 Incomplete Software Maintenance Requirements	13
2.4 Staffing	14
2.4.1 Absence of Software Architects	14
2.4.2 Absence of Software Specialists	15
2.4.3 Absence of Chief Software Engineer	15
2.4.4 Lack of Software Engineering Expertise for System-Level Requirements	15
2.4.5 Deficient Inter-Team Communication	16
2.5 Tools/Data Access	17
2.5.1 Access Limitations on the Concept of Operations	17
2.5.2 Requirements Traceability Tool Coordination	17
2.5.3 Collaboration Environment	18
3 Conclusions	20
References/Bibliography	21

Acknowledgments

The authors have been privileged to work with Peter Capell, Charles (Bud) Hammons, Ronald Kohl, Thomas Merendino, and Colleen Regan in support of a large, multi-segment software-intensive system for several years. This engagement is the impetus for the observations and potential lessons learned contained in this technical note. The authors are greatly indebted to the aforementioned teammates at the Software Engineering Institute for their insights.

Note: The views expressed in this paper are those of the authors and their colleagues.

Abstract

The Carnegie Mellon[®] Software Engineering Institute (SEI) supported a large, multi-segment, software-intensive system program for six years. During this time, the SEI team observed technical, organizational, and management situations that affected the overall execution of the program. This paper records some of the observations and describes suggested (and, in some cases, implemented) solutions—for both historical purposes and for their potential benefit to future programs.

1 Introduction

This paper contains a series of observations and their associated lessons learned from a large, multi-segment, software-intensive system. The system consisted of separately contracted multiple segments, with a separately contracted system engineering and integration (SE&I) team expected to lead integration and test efforts.

The Carnegie Mellon[®] Software Engineering Institute (SEI) supported this program for six years. During this time, the SEI team observed many situations from which the program could learn and/or future programs could learn. Some items cited may not have been directly addressed by the program, but should be acknowledged as program risks. Thus, any program that sees similarities should immediately recognize these situations as potential program risks.

There has been no attempt to document all lessons learned from the program; instead we discuss some that affect software or software-related topics. In addition, some of the topics address how larger overarching program issues impacted software development. This paper may be of interest to any program that can be considered software intensive.

Two definitions for software-intensive systems that could be useful to the reader are:

1. A software-intensive system is one that relies on software to provide core or priority mission function(s). A large software-intensive system is one whose software meets one or more of these criteria:
 - takes longer than six months to implement
 - takes more than six people to implement
 - takes more than \$750,000 to implement
 - comprises multiple, interrelated systems or independently developed components implemented in software (system of systems, family of systems, and the like)[Albert 2003]
2. A software-intensive system is one in which software represents the largest segment in one or more of the following criteria:
 - system development cost
 - system development risk
 - system functionality, or development time[DAU 2005]

1.1 Organization of the Paper

This paper is divided into three sections:

- Introduction (this section)
- Observations and Associated Lessons Learned
- Conclusion

[®] Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

1.2 Observations Content

The observations are loosely partitioned into five topics covering 19 individual observations and associated lessons learned. They are

- acquisition life cycle
- architecture
- requirements
- staffing
- tools/data access

Within each topic, several observations are addressed. Each observation follows the following format:

- Observation—a short statement of the observation
- Software Impact/Consequence—a description of the issue observed.
- Lesson Learned—what was or could be learned from the observation.

2 Observations and Associated Lessons Learned

The large, multi-segment, software-intensive system was characterized by multiple segments and system-level software issues, constraints and complexities. Many of the software techniques and practices used on this program were best practice and should be employed by other programs.

Among these items are the following:

- The program instantiated a system-level software integrated product team (IPT) that worked with the software personnel from each segment. The members of the software IPT had sufficient expertise to provide insight and advice that would make a difference for the software acquisition.
- The program office recognized early on the need for a system-level information model and sanctioned the development of such. While never fully completed, great strides were made in setting up a baseline from which to build.
- Senior leadership realized that software assurance had potentially large implications for the software and the overall system. They sponsored a system-wide symposium on software assurance which led to further focus groups that addressed software assurance specifics.
- The segment contractors specified a methodology/procedure for the design teams to transition system/segment design artifacts into the world of software artifacts while remaining consistent and traceable. In effect, they found a way to track DODAF type artifacts to software artifacts.
- The software IPT established a complete and comprehensive set of software measurements early (pre-Milestone B) in the program. These measurements were documented in the system-level Software Acquisition Management Plan.

The remainder of this section addresses observations, associated lessons learned, and provides suggestions for additional valuable practices from which the program might have benefited. The five topics are

- acquisition life cycle
- architecture
- requirements
- staffing
- tools/data access

2.1 Acquisition Life Cycle

2.1.1 Increase Number of Interim Design Reviews (IDR) and System Capability Reviews (SCR)

2.1.1.1 Observation

At the system level, the program encountered considerable indecision when discussing or documenting segment edge-to-edge versus *system* end-to-end behaviors. Edge-to-edge behavior is concerned with specifying interactions and activities that occur only within a single segment. End-to-end behavior includes specifying related interactions and activities that occur across the entire system, thus including multiple segments.

The “critical” quality attributes for reliability, fault tolerance, safety, and system quality are not well defined for the end-to-end system. There is some definition for each segment (edge-to-edge) but it is not self-evident how these definitions are meant to extrapolate and apply to the entire system. Additionally, these definitions are not necessarily described consistently for each segment. Inadequate specification and understanding of the system-level quality attributes is one example of system- or enterprise-level issues within the program.

Since the overall system-level qualities were neither adequately defined nor well understood, these system- or segment-level issues ended up being poorly addressed during formal program system or segment design reviews. This further resulted in considerable design indecision upon exiting the formal reviews. The program team’s integrated master schedule shows several years are planned to elapse between such scheduled formal reviews. Several years are far too long to wait to discover that these kinds of system- and enterprise-level issues exist or to assess the programmatic effects of indecision.

One method the government could employ to alleviate some of the system issues is to consider mandating an interim design review (IDR) or system capability review (SCR) every six months (but no longer than 18 months apart), as appropriate. The intent of either of these two types of meetings is to understand, sooner, how the status of these system or enterprise-level issues has changed, rather than waiting between the long separations of the formal acquisition review milestones. If the system program office determines that formal reviews such as IDR or SCR would be too costly to hold more frequently than 18 months, technical interchange meetings (TIMs) could be held every six months to ensure the continued discussion, resolution, and status awareness of system- or enterprise-level issues.

2.1.1.2 Software Impact/Consequence

Major acquisition programs often find they need to put a stake in the ground periodically to reduce the swirl of indecision that may emerge from the segment (that is, contractor) and system PMRs. The suggestion might be to set up IDRs or SCRs at the midpoint between each major development milestone (that is, midway between each of the intervals SRR–SDR, SDR–PDR, PDR–CDR, and CDR–TRR) for each segment; or to conduct the reviews on a regular interval of, say, six to 18 months.

The decision to use IDRs and SCRs rests solely with the system program office, not the segment contractors. The intent *is to ensure the government understands how each segment plans to inte-*

grate as well as sees where decisions are made for an edge-to-edge (segment) solution versus an end-to-end (system) solution. The government in turn must ensure the contractors are assigned tasks to support these IDRs and SCRs.

The challenge for the system program office is to determine *when* to put a stake in the ground so that the enterprise- or system-level review can present the consequences of design decisions at the segment level that are causing system-level problems. In addition, incremental use of this type of review will help mitigate risk and decrease the chance of declaring a Nunn-McCurdy or Levin-McCain breach.

One of the topics at these IDR and SCR reviews should be architecture and its associated critical quality attributes, such as high reliability, fault tolerance, safety, and system quality. These topics need to be important at every review so the contractors can show increased detail in their contract data requirements lists (CDRLs) over time. The timing of CDRLs will need to change to include deliveries of draft versions for fault analysis, positioning, signal integrity, and continuity analysis (PSICA), and hardware-software interface (HSI) analyses as appropriate earlier in the development cycle. This presumes the system program office has the resources and availability to respond to these documents in a reasonable time so that the contractor may respond to deficiencies and shortcomings in a reasonable timeframe. An example of this approach follows:

- at the SDR, include delivery of draft CDRL versions of fault, PSICA, and HSI analyses
- include delivery of updated draft versions of each analyses CDRL at the PDR
- include delivery of final draft versions of each analyses CDRL at CDR
- include delivery of the final version of each analysis CDRL at test readiness review (TRR)

If draft versions of these analysis CDRLs are made available by SDR, the IDR or SCR that is proposed to occur between SDR and PDR could require delivery of draft versions for use case and mission thread documents. Then, it seems reasonable to require delivery of the final draft version of the use case and mission thread documents between PDR and CDR. This approach would force delivery of these documents earlier in the program life cycle, thus ensuring the off-nominal paths are defined before CDR. Defining off-nominal paths is important on complex systems in order to promote making earlier design decisions needed to address off-nominal path system behavior.

The suggested approach would require changes to the existing segment contracts.

2.1.1.3 Lesson Learned

Gaining access to the design information incrementally will help to ensure that fault management, safety, and human factor concerns are factored into the design early on. Additionally, off-nominal paths will be defined earlier in the overall life cycle.

2.1.2 Capability Based Testing is “Test as You Fly ...” Testing

2.1.2.1 Observation

In this post-acquisition reform period, some “back to basics” is needed. The SE&I contractor and government-planned tests must prove basic *operational* requirements. The SE&I contractor and government need to recognize that tests conducted at the segment level, although proof of the

contractor *design*, will not necessarily prove the required operational behavior across all segments (that is, the system level) from the real end user perspective.

2.1.2.2 Software Impact/Consequence

Sufficient testing must be undertaken to prove the system operates as intended by the users, and not just as intended by the developers. The final system must meet the intentions of the end users, both the system operators and the users in the field. During the 1980s and early 1990s, government test concepts mandated tests that proved the real normal operations used and planned by the operator. In addition to testing and proving the system, these tests could help correct workflow or operator procedures that need to change with new system changes. The tough tests were the operational procedures when off-nominal operations were being experienced—which became the stress test of the software.

Currently the model is for the government acquisition organization to work closely with a third-party government test organization. The combined development test and operational test (DT/OT) period needs to be operationally focused on testing the end-to-end system. *DT/OT must not simply accept the contractor's testing of segments and conclude, by inference, that just because the pieces worked, that the entire system will also work properly.* Normally, inter-segment software operation is proven through integration testing, not assumed by inference.

The bottom line of “test as you fly” is to ensure the system works as intended. This is one of the tenets of software assurance.

2.1.2.3 Lesson Learned

The government must be responsible for testing the composite of the system to ensure the system operates as intended. In other words, software assurance requirements are an integral part of the system test.

2.1.3 Inadequate Configuration Management

2.1.3.1 Observation

The program needs a more global approach to configuration management (CM) throughout the entire acquisition life cycle.

2.1.3.2 Software Impact/Consequence

The program acquisition strategy led the program office to initially confine its CM activities to management of the government-side specification tree. Additionally, the system program office assumed that contractor-specific CM practices would be sufficient to govern CM activities in the segments. The problem occurred when system-level technical documents were created, such as system test documents, system configurations, and the like. The government-side specification tree was not set up to accommodate these documents and the segment contractors were not accountable as these were not segment documents—they were system documents. A more thorough approach would have been comprehensive CM

- for all program artifacts

- marking conventions to address configuration compatibility across segment boundaries (for example, declared operational configuration for any configuration of the entire program)

2.1.3.3 Lesson Learned

The inadequate scope of the CM approach employed by the program office allowed more entropy to creep into the organization of the program artifacts—an undesirable amount. The absence of constraints on intersegment configuration marking tends to force manual processes in operations that could be computer-assisted or partially automated.

2.1.4 Acquisition Approach

2.1.4.1 Observation

Staggered start of segments created problems with identifying and addressing systems-wide software issues.¹

2.1.4.2 Software Impact/Consequences

Due to the staggered start, software decisions made by the first contracted segment were becoming default standards that would eventually be imposed on the yet-to-be-let segments. In the future, if the newest segment under contract did not agree with these default standards, negotiations between the segments would be needed to reach mutually agreeable standards. This would affect schedule at the very least and most likely cause some rework for the first segment. In addition, necessary software decisions might be (or were) deferred due to the later contractor, not yet chosen.

2.1.4.3 Lesson Learned

To overcome this challenge, strong software leadership with a program-wide perspective should have been installed early in the project and given authority to influence software decisions just as other chief engineers have control and influence over other aspects of the program (e.g. hardware, network, and the like).

2.1.5 Disregard for System-Level Software Considerations

2.1.5.1 Observation

Early in the program (pre-Milestone B), an independent evaluation of the program was held. One of the findings was that a system-level software architecture was missing. There were varying opinions from the program office leadership team about whether this artifact was indeed necessary, since the overall program approach was that software was to be developed at the segment level. Ultimately, the leadership decided to create a system-level software architecture artifact.

2.1.5.2 Software Impact/Consequence

Systems-level decisions were made in what appeared to be the absence of systems software engineering subject matter experts and accompanying system-level software artifacts. These decisions profoundly influenced and constrained downstream software decisions and options. One visible consequence of this perspective was the absence of *any* software-related critical technology ele-

¹ The authors did not have the opportunity to observe issues with all the segments.

ments (CTEs) identified during the program’s technology readiness assessments (TRAs). Given the projected size of the program—multiple million lines of code—no software-related CTEs is something of a flashing warning light.

The resulting unintended consequences to software are likely to include inadequate attention paid to global architectural and testing issues, creation of additional code to address deficiency work-arounds, missed integration schedules, and associated cost growth as unintended issues pop up late in the acquisition, when they are *much* harder to correct. Identifying appropriate software-related CTEs may help focus attention to address risky areas in software sufficiently early in the program life cycle.

2.1.5.3 Lesson Learned

While the program office leadership was aware of and tacitly supported software issues, active production of software artifacts at the system level was needed to reinforce that support. The old adage “actions speak louder than words” seemed to apply here. By taking actions to produce system-level software artifacts, the senior leadership would signal (and thus educate) their staff that software is an important, indeed vital component of their system. Without such senior-level support, software decisions and accompanying artifacts may not occur in a timely manner, thus inadvertently impacting the overall program as stated above.

Specifically for CTEs in a software-intensive program, continue pressure to locate justification (from the program’s TRA support team) that addresses why no software CTEs were identified and whether the TRA’s independent review panel (IRP) had queried this issue. Flag this issue to program office leadership should the credibility of the justification seem weak. Since the historical track record for delivering large DoD software-intensive programs on-time and in-budget is less than stellar (many times, due to software issues), this issue likely becomes a significant assumed risk on the government’s part, even though the risk may not be formally acknowledged to exist.

2.1.6 Software Proposal Evaluation

2.1.6.1 Observation

Proposal evaluation can result in a sense that strong software leadership is lacking in the offering organization. If an evaluator does get this sense, then they should look deeper at the software approach.

2.1.6.2 Software Impact/Consequence

When evaluating proposed approaches to developing and integrating software, it is tempting to accept, at face value, the offeror’s description of roles and responsibilities for its key leadership positions. It is also tempting to accept apparently good documentation of development processes as a supplement to software leadership accountability. However, all such descriptions need to be analyzed in total to assess whether software leadership and accountability roles are clearly observable. If such analysis does not result in absolute clarity of strong, overall software leadership and accountability, it may signal an underlying weakness in the offeror’s overall software—and perhaps system—approach.

2.1.6.3 Lesson Learned

It is easy to become overly focused in the details describing the offering organization's key position descriptions as well as with the descriptions detailing the strength of its established processes. While these are necessary components of achieving success in developing software-intensive systems, they are likely not sufficient.

The evaluator must always be mindful of the end goal—given the offeror's approach to software, will the quality and executability of the program be satisfactory? The following additional questions and criteria are suggested as a vehicle to provide further, detailed insight into the offeror's approach:

- Is there a chief software engineer, and what is this person's role?
- Is there a chief software architect, and what is this person's role?
- How are these positions distinct? How do they overlap? What is the scope of authority of each?
- How do these positions interact with the chief systems engineer? Other chief engineers?
- What is the defined power and influence of the chief systems architect and engineer over the software systems that need to be created?
- Is software a “full, equal partner” in the overall development approach? Or is it delayed and deferred?
- To what extent is software considered during system decisions of alternatives? What supporting documentation of such considerations exists?
- How does software fit into the described processes for the system's life cycle? Explain the significance of the software component.
- How do software engineering activities fit into the system engineering activities in a software-intensive system with millions of lines of code (LOC)?

2.2 Architecture

2.2.1 Administrative Decomposition of the System has Sweeping Technical Effects

2.2.1.1 Observation

Conway's Law is an architectural concept which suggests that the system functional decomposition will mirror the administrative structure employed to implement it [Conway 1968]. This program was no exception in that the overall structure of the system mirrored the system's division into its largest organizational chunks—segments. This particular division of the system into a segment structure did not appear to take into account any of the software boundaries or the complex interfaces created by the forecast six million LOC.

2.2.1.2 Software Impact/Consequence

While the term “stovepiping” provides some of the sense of the consequence of the division of the program, it is inadequate to describe the problems of communication, coordination, and technical management that were induced with the creation of multiple, co-equal, system-level actors where none were bound by any requirement to interoperate with the others. This division was enough to

create instability in most aspects of the conduct of the program; however, having no singular authority and no single systems architecture representation allowed for the independent entities to call their own shots, irrespective of the need to collaborate intensely to achieve program goals. From a software perspective the early partitioning and allocation decisions could and did have dramatic impact on the complexity of the software. In many cases, this caused complex interfaces to be created and then managed.

2.2.1.3 Lesson Learned

The impacts of creating a stovepipe-type organization need to be addressed when the program is formed. In particular, if such an organization is created either within the system program office or within the contractor organization, strong requirements for interoperability and inter-coordination need to be identified early and managed throughout the program. These interoperability and inter-coordination issues are paramount for the software areas. For instance, it becomes essential that an information model be completed early in the program and then rigorously enforced throughout the development and life of the program in order to ease the development and integration of software across the segments.

2.2.2 Inadequate Software Architecture Focus

2.2.2.1 Observation

A list of computer software configuration items (CSCIs) is emphasized as the primary, or only, representation of the high-level system software architecture. This is inadequate, given the following definition of software architecture:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003].

A list of CSCIs only addresses the structure, but not the externally visible properties nor the relationships among them.

2.2.2.2 Software Impact/Consequence

This perspective tends to result in development and acquisition team structures being organized around the CSCI structure. Since the CSCI structure generally results from functional decomposition analysis, the resultant team structure mimics the functional decomposition. Without very strong technical leadership and oversight, teams organized in such a manner tend to focus in isolation on their allocated functions. This may distract teams from emphasizing the development of holistic behavioral and quality aspects that are really important to system stakeholders, resulting in the belated realization at integration time that important qualities are absent.

This situation is likely to be observed in development contractors that have strong emphasis on processes based in traditional systems engineering methods. There is likely to be a strong functional decomposition flavor in the contractor's approach to the system design. Typically, a harbinger for this perspective is the initial existence of a SEIT organization (systems engineering, integration, and test) that drives the front part of the development life cycle using primarily (or only) a functional decomposition approach.

System software architecture is a key contributor to a system's ability to meet stakeholder expectations of cross-cutting system quality attributes (or non-functional requirements) such as flexibility, maintainability, availability, performance, security, etc. If SEIT decomposition carves up functionality without early, in-depth, and iterative consideration given to the resultant impacts on the complexity of software architecture, the expected levels of cross-cutting quality attributes, as well as unanticipated software interactions, will not likely be addressed until system integration time. This places the program at high risk of exceeding cost and schedule during the system integration phase. Indeed, it may not be affordable to repair insufficient system quality attributes during system integration time, since many of these qualities are a result of fundamental architectural design decisions made long before.

2.2.2.3 Lesson Learned

A key lesson learned here is that introduction of collaboration activities *after* the organization of development teams and IPTs should be carefully examined to understand whether they actually represent a real process improvement, or are an attempt to fill a gap due to a lack of holistic understanding of how systems or software architecture impacts resultant system quality attributes. Oversight of this collaboration needs to be led by government personnel who have sufficient experience and knowledge in software development and acquisition. The aforementioned experience and knowledge allows the government to fully understand the imminent software hazards and threats so that they can be addressed in a timely manner.

A second lesson learned is to beware of how the organization of the execution teams may isolate the development of individual pieces of the system, resulting in lack of adequate consideration being given to the overall system non-functional requirements and desired operational behaviors.

A third lesson learned is that the system software architecture is important to communicate among stakeholders, document early design decisions, and transfer abstraction to the system [Bass 2003]. In addition, "the right software architecture is essential for a software-intensive system to meet its functional requirements as well as its quality requirements that govern real-time performance, reliability, maintainability, and a host of other quality attributes. Because an architecture comprises the earliest, most important, and most far-reaching design decisions, it is important for an acquisition organization to exercise oversight prerogative with respect to software architecture [Bergey 2005]." There is an entire body of knowledge that addresses the need for and how to develop software architecture but that is beyond the scope of this paper. A search of the SEI website for system software architecture is a good starting point.

2.3 Requirements

2.3.1 Frequent Requirement Change Caused Significant Rework and Analysis Churn

2.3.1.1 Observation

Segment leads have stated in test meetings that the level of requirement change every six months affects the program's ability to progress.

2.3.1.2 Software Impact/Consequence

Every program should expect some requirements change. For example, many operational organizations have a major board that reviews new requirements at least annually for validation. Program offices need to be prepared for this requirement review. In turn, the program office and contractors need to understand that a new requirement is not a reason to re-plan the entire program. The program office and the operational organization realize these new requirements can introduce engineering change proposal(s) (ECPs) along with appropriate funding adjustments, but should not expect a program re-plan. On this large-scale software-intensive program, the program office and its support staff allowed this “swirl” of program re-plan. This “swirl” impacted everything, including software, hardware, integration, etc.

As an example of what has happened, the program experienced several years of activities with multiple versions of the technical requirement document (TRD). Each version of the TRD seemed to allow a program re-plan that, in turn, halted progress because requirement analysis and requirement mapping needed to occur to determine impact on the program. This is not a suggestion to drop the requirement traceability, but to use or define the requirements traceability tool attributes to store requirement analysis or gap analysis and trade-off decisions. This stored attribute will help sustainers and maintenance organizations understand options considered and dropped due to funding or adverse impact on design complexity for both hardware and software. The sustainment group needs to understand how the program changed over time; see how thoroughly ideas or problems are captured; and see the results of the analysis in terms of design and test documents (including test results).

The contractor potentially sees each new requirement as either a new way to increase their contract value via an ECP or an opportunity to rework all requirements to show that the program is no longer possible within the current cost value. Input from the government segment leads suggested real engineering work is hampered because the contractor is allowed to focus on refactoring the requirements, determining design impacts, and generating the document churn mandated by the contract.

Instead of allowing the contractor to focus on refactoring the requirements, the government needs to enter the document review and comment resolution negotiation as soon as possible. This will allow the program to meet its deliverable objectives instead of turning into a requirement churn exercise.

2.3.1.3 Lesson Learned

If a program was prepared to address new requirements on a regulated interval, the planning by the program management might be able to pre-plan the activity and determine ways to reduce the volume of time and churn related to changing requirements. It is essential that the requirements become stable so that initial architecture and top level design for the system including software can be accomplished. In addition, the program needs to establish techniques to record changes to the requirements over time for historical, maintenance, and sustainment purposes. This record of requirements actions is extremely important for software developers so they can understand what was tried and rejected and why.

2.3.2 Error Conditions Not Completely Defined

2.3.2.1 Observation

Requirements that identify faults and failures that the system will detect, inform or alert users about what to do, and provide recovery functions are incomplete. The faults and failures represent the off-nominal behavior for the program.

2.3.2.2 Software Impact/Consequence

The lack of a complete set of requirements for error conditions impacts the ability of the system to satisfy its user needs and to allow testers to develop comprehensive tests. Typically, the off-nominal requirements are implemented with the software. Without a complete set of requirements, the error handling functions will be incomplete. This can lead to an under-estimation of total cost, schedule, and size (e.g., software sizing), and major surprises during integration and test.

2.3.2.3 Lesson Learned

To overcome this challenge, a set of fault and hazard analyses needs to be performed early in the life cycle of the program to determine which faults should be dealt with and the size of the corresponding functionality (software, hardware, procedures, and so forth) to deal with those faults. Paragraph 2.1.1.2 provides some additional insight as to when and what type of input could be provided early in a program. Typically, programs should avoid writing only use cases and requirements that emphasize the “happy path” for the system with no regard to off-nominal or exception paths.

Common objections to this off-nominal approach are that it is too early to do off-nominal analysis or that it is design specific. We do not find these arguments compelling. There are off-nominal paths that can be identified in the operational architecture as well as the functional decomposition, both of which should be technology and design independent. Capturing these off-nominal paths early on in the program sets the stage to add design-specific off-nominal behavior without confusing it with off-nominal behavior that is inherent in the missions to be accomplished or the high-level functionality of the system in question.

2.3.3 Incomplete Software Maintenance Requirements

2.3.3.1 Observation

Incomplete software maintenance requirements will lead to inadequate system functions to support the full range of software maintenance needs.

2.3.3.2 Software Impact/Consequence

Incomplete software maintenance requirements can lead to new requirements introduced late in the development life cycle and to inadequate operational capabilities once the system goes into operation.

2.3.3.3 Lesson Learned

To overcome this challenge, it is important to have software-savvy folks involved in early system-level requirements development. This particular topic is often assigned to the logistics or specialty engineering teams, which often do not have the necessary software skills. In many cases, the logistics and specialty engineering personnel planned for the software maintenance capabilities but cannot be sure all needs are being met.

2.4 Staffing

2.4.1 Absence of Software Architects

2.4.1.1 Observation

Absence of software architects during synthesis of system requirements leads to requirements and associated solutions that do not consider software capabilities, implications, or tradeoffs.

2.4.1.2 Software Impact/ Consequence

Current system engineering practice is to defer software considerations until after requirements are synthesized and allocated, so it is customary that software architects or other software specialists have limited to zero influence on system requirements. Unfortunately, this was also true on this effort. The lead software person in the program office was for the most part not included in the initial synthesis and allocation of software requirements. This gap was ultimately recognized by the program office, which approved the incorporation of system-level software architecture constraints in version 11 of the system TRD. This action was necessary to recognize the influence of the architectural constraints imposed by the Net Centric Key Performance Parameter (KPP).

At the system level, there should be interrelated high-level software design decisions and trade-offs being made in a coordinated manner, as well as multiple interactions between the segments and external elements so that the desired system capabilities and behaviors are synthesized. Given this, it would be misguided to believe that the program can subsequently synthesize desired system-level capabilities and behaviors by simply allocating software to just the segments without truly understanding that some system-desired-level capability could be lost (i.e., some important and expected system-level characteristics could be lost). Conversely, there are early partitioning and allocation decisions that have dramatic impact on the complexity of software, which should be vetted by people experienced in software architecture and software development.

Early involvement of software architects and experienced software developers will increase the probability that the government can prevent downstream software issues whose origins lie in early program level decisions. At the very least, with system-level definitions of software and related items, misconceptions and disconnects are less likely, and the need for corresponding, interrelated segment-level software design decisions to synthesize desired system characteristics may be more reliably identified and those decisions taken.

2.4.1.3 Lesson Learned

Absence or deferral of software architects involvement during synthesis of system requirements leads to a set of system requirements that do not include adequate treatment of quality attributes or life-cycle evolution issues that uniquely affect software, as it is the most malleable portion of

system implementation. This in turn leads to a system architecture that is created without consideration of tradeoffs and impacts of software to the system.

2.4.2 Absence of Software Specialists

2.4.2.1 Observation

Absence of software specialists during interface development leads to inadequate interface definitions.

2.4.2.2 Software Impact/Consequence

Interfaces are often simply a collection of messages and data formats, not reflecting the complexity of interactions among segments and elements. This model contributes to lowest-common-denominator interface specifications that do not capture behavioral dependencies of interest to software developers. The behavioral dependencies could be expressed as stateful protocols or use cases reflecting a “conversation” occurring across an interface.

2.4.2.3 Lesson Learned

Require that software engineers and architects are part of the interface development teams so that behavioral dependencies between software and emergent software properties can be addressed during the creation of the interface definitions.

2.4.3 Absence of Chief Software Engineer

2.4.3.1 Observation

All programs developing software, especially those consisting of over one million LOC should, at program initiation, have a chief software engineer; said individual should be equivalent in stature to the program’s other chief engineers.

2.4.3.2 Software Impact/Consequence

Software is a critical component of systems. By ignoring software or not considering the impacts or implications that system engineering and system architectural decisions have on software, the program or system will encounter significant integration, cost, and schedule issues.

2.4.3.3 Lesson Learned

By proactively including influential software personnel from the early stages of a program, the system engineering and architectural decisions will include a complete view of the whole program and potential impacts, both hardware and software. This will allow the program to address software engineering issues early and provide mitigation or prevention of issues that many times do not appear until late in the program, usually during integration. Thus, quality is improved while additional cost and schedule overruns are avoided.

2.4.4 Lack of Software Engineering Expertise for System-Level Requirements

2.4.4.1 Observation

There was a lack of software engineering expertise involvement in defining and developing the initial systems-level requirements for software capabilities (such as software maintenance, software security, error handling for software errors, and the like).

2.4.4.2 Software Impact/Consequence

Without adequate software engineering expertise involved in defining system-level requirements that relate to operational software capabilities, it is likely that system requirements will be inadequate. This can lead to underestimated system and software size estimates, inadequate systems-level software architecture, incomplete operational capabilities related to software, and impacts on systems testing and operator training.

The initial versions of the system requirements contained some inadequacies in the areas of software maintenance, software security, and software error handling. The software IPT did become involved in later versions of the requirements. This involvement included a thorough review of the system requirements from a software focus. Ultimately, suggestions were made and presented to the program office in the form of briefings and papers that provided solutions for any inherent weaknesses.

2.4.4.3 Lesson Learned

The acquisition office needs to ensure that adequate software expertise is involved at the beginning in the development of requirements for operational capabilities that relate directly to software. This should involve software representation from the acquisition and contractor perspective and also from the operational perspective (such as a responsible software maintenance organization once system goes operational). Earliest possible involvement by software personnel can help avoid potential rework which impacts schedule and budget.

2.4.5 Deficient Inter-Team Communication

2.4.5.1 Observation

Contractor teams exhibited visible evidence of turf battles and deficient inter-team communication.

2.4.5.2 Software Impact/Consequence

A contractor's development organization, as a single company, is likely guided by consistent company-wide goals. Typically, on large programs badgeless environments are created to encourage open communications both within contractor groups and with the government. In spite of this, all of the various teams still may not "play nicely in the sandbox" due to personalities, personal biases and local motivations or incentives. Such biases or motivations, should they exist among different parts of the development organization, are likely to cause inefficiencies and poor communications within the contractor team. Collaboration and close communication among the different teams within a contractor's organization is crucial for successful development of any complex system. This is especially crucial for developing complex software that must ultimately be integrated and interoperable. Unabated, this type of activity forces the government program to absorb the cost of the inefficiencies resulting from deficient contractor inter-team communications.

2.4.5.3 Lesson Learned

Developing systems with complex software typically involves intense coordination among many different software development teams and are thus generally high stress environments. It is nor-

mal to expect personal and team disagreements will occur. However, it is also important to monitor inter-team coordination, and recognize and “flag” when the levels of personal or team disagreements within a contracting organization are rising to levels that are likely to impact their effectiveness. If such activity is recognized, the government needs to hold open and frank discussion with the contractors in an effort to resolve the issues. If this does not work, then the government may need to act through award fees and other contractual mechanisms. Otherwise, the government may be assuming a risk that perhaps should be formally recognized and mitigated

2.5 Tools/Data Access

2.5.1 Access Limitations on the Concept of Operations

2.5.1.1 Observation

Initial tight control of the concept of operations (CONOPS) access may be needed due to the government still defining the “target of opportunity” and ensuring there is no competitive advantage given to one contractor over another. By year three, four, or five of the acquisition, the CONOPS should be sufficiently stable that the government could widen distribution of the CONOPS unclassified and classified material.

2.5.1.2 Software Impact/Consequence

Many of the active participants on the program never had access to the CONOPS and thus were designing in a vacuum. Giving the government’s view of operations to all contractors would help to ensure a better understanding of the system’s or enterprise’s operational environment. There is a tendency to let the first segment lead the way, and make design decisions for the system or enterprise that are not always good for the system or the enterprise. One segment should not have the ability to force its design decisions onto another segment.

2.5.1.3 Lesson Learned

Access to at least the unclassified portion of the operational concept would help focus all segment teams on the operational environment and the end-to-end perspective of the system. As the contracts are let, the segment contractor needs immediate access to determine the design impacts of the more detailed operational, user view.

2.5.2 Requirements Traceability Tool Coordination

2.5.2.1 Observation

On this program, the government spent an inordinately high amount of time and money to enhance the requirements traceability tool (such as DOORS), so that it can have the requirement linkages to testing information (test plans, procedures, automated test scripts, etc.). The cost to purchase a product like HP-Mercury Interactive requirement management tool, and to pay for the tool vendor services to tailor it to a program, is less costly than having the government, SE&I, and each segment contractor trying to set up a flat-file database structure to look and act like an object oriented or relational database.

2.5.2.2 Software Impact/Consequence

The test attributes feature being added to the requirements traceability tool was worked and re-worked by the test and verification IPT within SE&I for more than one year. The team working this effort expands and contracts, but the level of effort far exceeds one person-year. The cost expended to define what is needed is so high that it remains unclear why the approach did not change to purchase “Tool X,” develop the import and export functions for the requirements traceability tool to maintain the requirement linkages, and develop the reporting needed to meet each working group and IPT requirement need. The requirement management tools that include test automation have some prepackaged reports but with a few training sessions the SE&I contractor could program and maintain the reporting component of these tools. This overarching issue impacted the availability of the requirement information to all disciplines including software. While this could be considered a minor issue, it does impact the requirements phase of the program by limiting data access and coordination across various teams. For the software team, it made the task of reviewing and coordinating on various aspects of the requirements time-consuming and difficult.

2.5.2.3 Lesson Learned

The government needs to realize the limits of its requirements traceability tool. In addition, the government needs to understand that, although the cost for a tool such as HP-Mercury Interactive seems high at the beginning of a program, when the tools are in place early the overall sustainment cost of the system will be reduced. When a program elects to transition to automated test tools, it will find some rework introduced to set up the linkage to requirements from the automated test scripts, test plans, procedures, and the like. If the linkage between requirements traceability tool and the tool of choice is completed by or with the tool vendor it will be less expensive than having your SE&I or segment contractor struggle through that development. Each contractor will need to maintain the tool queries, reporting, and other factors—but they will want training from the tool vendor to define the level of work needed to complete the initial requirement mapping, and then to maintain the requirement mapping. Once there is buy-in into the vendor tool, progress on the program is not hampered by the tool and progress improves as the system design gains complexity.

2.5.3 Collaboration Environment

2.5.3.1 Observation

The program used a collaboration and content management environment that was called an integrated development environment (IDE). This conventional collaboration and content management environment proved to be inadequate for distributed development teams such as seen on this program. Typically, anyone on the program with access to the IDE could and did post data. This posted data was meant to be shared and used for collaboration with others on the program. There appeared to be few if any rules, other than enforcing organizational conflict of interest (OCI) rules. Thus, using IDE to find data became very cumbersome at best. Team members charged with software-specific roles were particularly vulnerable to this shortcoming, because software artifacts were not centrally managed, but allocated across the segments and the various elements. For software issues that cut across segments, this meant a lot of searching to find relevant artifacts was required.

2.5.3.2 Software Impact/Consequence

Conventional IDEs are functionally equivalent to file cabinets. For this program, the drawers within the file cabinet were labeled according to the program's OCI rules. Thus, there was a drawer for contractor A, a drawer for contractor B, a drawer for contractor C, and so forth. There did not appear to be any overall organization schema for the use of these "drawers." If one did exist, it was not well advertised. Thus the level of organization is heavily contingent upon the degree of discipline of the user community.

The existing approach for IDE organization tended to be distinct for each group, and each group tended to operate within independent stovepipes. The current IDE approach was a major impediment to the dissemination of information to stakeholders, managers, acquisition personnel, and developers. The problem for software specialists is that software content in each area was described and/or alluded to in divergent locations and employed nomenclature peculiar to the working group or IPT in charge of each specific area.

2.5.3.3 Lesson Learned

Before putting a program artifact and data repository into use, an overall document architecture should be created that can be replicated across all program entities. Forgoing a disciplined and overarching programmatic approach for using available search capabilities is not an appropriate option. The overall artifact architecture would help users to find information faster and easier. In addition, a website managed by IT specialists and an information librarian would be preferable. These personnel could devise a thorough artifact architecture complete with a configuration model with graphically based navigation aides to map system constructs to artifact storage. Naturally, this is an additional cost that the program office may choose to avoid, but the benefits will probably outweigh the costs. In programs where software development responsibility is aligned with the physical decomposition of the system, a consistent approach to the program repository will allow more efficient and more reliable access to relevant artifacts.

3 Conclusions

This paper presented some observations and lessons learned, as observed by the SEI team during our support of a large, multi-segment, software-intensive system. All of the material described in this paper affected the process of developing software and its associated artifacts. It is the hope of the authors that this material and experience can be applied to the acquisition and development of other large software-intensive systems.

References/Bibliography

URLs are valid as of the publication date of this document.

[Albert 2003]

Albert, Ceci; & Brownsword, Lisa. *Software Intensive Systems Briefing*. Software Engineering Institute, 2003.

[Bass 2003]

Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice Second Edition*. Addison-Wesley, 2003. www.sei.cmu.edu/library/abstracts/books/0321154959.cfm

[Bergey 2005]

Bergey, John K.; & Clements, Paul C. *Software Architecture in DoD Acquisition: A Reference Standard for a Software Architecture Document* (CMU/SEI-2005-TN-020). Software Engineering Institute, Carnegie Mellon University, 2005.
www.sei.cmu.edu/library/abstracts/reports/05tn020.cfm

[Conway 1968]

Conway, Melvin. *How Do Committees Invent?* F.D. Thompson Publications; reprinted by permission of *Datamation*, April 1968. www.melconway.com/research/committees.html

[DAU 2005]

Defense Acquisition University. *Glossary—Defense Acquisition Acronyms and Terms*, 12th ed. Defense Acquisition University Press, July 2005.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 2009	3. REPORT TYPE AND DATES COVERED Final		
4. TITLE AND SUBTITLE Lessons Learned from a Large, Multi-Segment, Software-Intensive System		5. FUNDING NUMBERS FA8721-05-C-0003		
6. AUTHOR(S) Mary Ann Lapham John Foreman				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2009-TN-013		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER		
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE		
13. ABSTRACT (MAXIMUM 200 WORDS) The Carnegie Mellon® Software Engineering Institute (SEI) supported a large, multi-segment, software-intensive system program for six years. During this time, the SEI team observed technical, organizational, and management situations that affected the overall execution of the program. This paper records some of the observations and describes suggested (and, in some cases, implemented) solutions—for both historical purposes and for their potential benefit to future programs.				
14. SUBJECT TERMS Software-intensive system, program office, acquisition, lessons learned		15. NUMBER OF PAGES 31		
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	