The Perils of Treating Software as a Specialty Engineering Discipline

Keith Korzec
Thomas Merendino

April 2013

During our support of various acquisition programs within the U.S. Department of Defense (DoD), the authors have observed that system development methods employed by acquisition program offices and by contractors tend to insufficiently engage key software domain experts during the initial synthesis of requirements and systems architectures. A key characteristic of utilizing such methods often results in a physical or hardware-centric design focus during the earliest phases of a program. We have observed programs encounter difficulties that we believe are attributable to design approaches that underemphasize software engineering concerns during the early formulation of system requirements and architecture. We have also observed specialty engineering disciplines (i.e., safety, security, reliability, etc.) receive similar treatment. The topic of our paper is certainly not new, but we continue to observe problematic reoccurrence as more and more systems are being acquired that increasingly rely on software to accomplish mission-critical goals.

We present our position and opinions on this topic for leadership consideration, primarily within the government program management community. We summarize a few examples of representative difficulties that we believe have been introduced by design approaches that under-represent software engineering considerations in the early life cycle program phases. We conclude with some recommendations to help achieve a closer coupling of design methods across the various engineering disciplines that we believe can assist in reducing the overall risks during acquisition and development of software-reliant systems.

INTRODUCTION

Historically, our experience with DoD programs that acquire software-reliant systems illustrates that software engineering concerns are often sources of difficulty during integration.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

This topic is certainly not new.[1,2] Our recent engagements supporting acquisition programs drive us to conclude that much improvement is still needed to efficiently develop the growing number of acquired systems that rely on greater amounts of software to accomplish mission critical goals.

OBSERVATIONS

Exclusion of software disciplines

A key factor underlying these difficulties may be related to the delayed participation and exclusion of people with software engineering architectural and development skills from the Systems Engineering Integration and Test (SEIT) team. Specific organizational structures vary from program to program. However, we have observed that software expertise generally resides in one or more integrated product teams (IPTs) other than the SEIT. This segregation typically exists within both the program office and contractor organizations. From this aspect, the software engineering discipline, to its detriment, operates in a manner similar to the specialty engineering disciplines in terms of relatively separated roles and delayed participation in the overall requirements, architecture and design of large systems.

The high level structure of the system/segment "nodes" tend to bear a marked resemblance to the organizational structure of the SEIT and product development IPTs, which is a common occurrence described by Conway's Law.[3] The term "nodes" is used here to mean a grouping of system hardware and software whose

---

[1] Grady Campbell. *Reconsidering the Role of Systems Engineering in DoD Software Problems.* SEI presentation. January 2004. http://www.sei.cmu.edu/library/abstracts/presentations/campbelljan2004.cfm (accessed March 2012)

[2] NDIA Task Group Report. *Top Software Engineering Issues within Department of Defense and Defense Industry*. September 2006. (See issue #2 in this report, excerpt of which is reproduced in the Appendix below)

[3] Melvin E. Conway. "How Do Committees Invent?" *Datamation Magazine*. F. D. Thompson Publications, April 1968. http://www.melconway.com/Home/Committees_Paper.html

implementation is intended to provide major piece(s) of functionality. The structure of a system's nodes is usually determined in conjunction with system functional decomposition decisions. These key decisions are typically made by the SEIT very early in an acquisition program's life cycle. In order to achieve total life-cycle system planning, the SEIT must include all engineering domains.

Design decisions not coordinated

By the time such a system decomposition structure is formed, many important system design decisions (intended and unintended, documented and undocumented) are made. Some of these design decisions are likely to be cross-cutting in nature, strongly suggesting that coordinated design decisions are needed across multiple IPTs to realize system intent and to avoid unintended deviations. For example, a satellite system decomposition that defines a space segment and a ground segment can cause functions that are common across both segments, such as data analysis, system health, common operating picture/situational awareness, mission planning, etc.to be split between IPTs. These decomposition decisions will be subsequently inherited (knowingly or unknowingly) by the IPTs as constraints on the development and implementation of their assigned nodes.

During system development activities, we typically observe that IPTs perform their work largely autonomously. Their focus is on implementing the contractual product requirements that have been "flowed" to their assigned system or segment nodes. Additionally, these teams are influenced by strong schedule-driven incentives that intensify the teams' internal development and delivery focus.

In this setting, cross-cutting design decisions, made autonomously by individual IPTs, can significantly impact other IPTs and/or the system as a whole. Typically, such interrelated decisions are discovered late into design activities. By the time these are discovered, budget, schedule and IPT egos often result in resistance to revisiting and coordinating design decisions with other IPTs.

Inter-IPT collaboration, cooperation and negotiation are often the primary means by which such cross-cutting concerns are addressed. We have observed a variety of ways in which specific programs manage the oversight of inter-IPT collaborations. For example, some programs appoint the SEIT to have these responsibilities. Others may assign these activities to a separate team of architects. At the opposite end of the management spectrum, others may decide not to recognize this as an issue. Absent a very strong authority having oversight of these collaborations, the process has been seen to result in IPT decisions that are in the best interests of one IPT but may not be optimal for the overall system. This increases difficulty in management and control of the ongoing system development and is exacerbated further in programs with larger numbers of IPTs.

Hardware equals system

Additionally, we often observe hardware and systems engineering disciplines treated as being synonymous within programs, whereas each is really its own unique domain specialty. As a result of this treatment, hardware engineering is considered the lead engineering domain involved with making early, important system architecture decisions, with all other domains relegated to a supporting role later on. However, systems architectures that adequately address the concerns of software engineering, as well as those from other engineering disciplines, are more likely to result when systems engineering activities are performed early in a program's life cycle using iterative and collaborative interactions among the various domains of expertise.[4,5] Additionally, it is worth noting that the Department of Defense's Technology Readiness Assessment (TRA) Deskbook[6] recognizes the importance of considering software in the system level context by defining both hardware and software Technology Readiness Levels (TRLs) as well as by emphasizing the importance for programs to identify software among the system's Critical Technology Elements (CTEs) during the TRA process.

Emphasis on developing code

We have also seen that the software design and coding activities are frequently viewed as being the bulk of software engineering activities needed within the development of a system. This would be analogous to saying that the design and production of the physical hardware are the only hardware engineering activities needed within the development of a system. For hardware, it is well understood that significant systems engineering, hardware architecture, design and prototyping efforts are needed before any component can be formally designed and produced. For software-reliant systems, however, as much if not more effort is needed to make key architecture decisions at the system level that directly impact and guide subsequent software development activities. Software needs the

---

[4] Grady Campbell. *Reconsidering the Role of Systems Engineering in DoD Software Problems.* SEI Presentation. January 2004. http://www.sei.cmu.edu/library/abstracts/presentations/campbelljan2004.cfm (accessed March 2012)

[5] Donald Firesmith, et al. *Method Framework for Engineering System Architectures.* Taylor & Francis, 2009

[6] Office of the Director, Defense Research & Engineering (DDR&E). *Technology Readiness Assessment Deskbook*, US Department of Defense. July 2009

same level of systems considerations afforded to hardware engineering before the "manufacturing" of code starts in earnest.

Program success depends on software

Further complicating the situation, systems that are reliant on millions of lines of software code will end up with a system where software becomes ubiquitous. Greater amounts of complex system behavior and risk mitigation are dependent on successfully developing and integrating software. It is our position that increased participation by key software engineers and architects during the early systems requirements and architecture activities is one of the key approaches to help reduce schedule and cost risks. Depending upon the program, software participation could begin as early as the Capabilities Based Assessment (CBA).

SOME EXAMPLES

In this section, we describe a few examples of representative difficulties that systems development teams encounter when following development methods that separate and/or delay engagement of software engineering domain experts during the early phases of system development.

It is of interest to the authors to note that such development methods frequently delay and/or separate engagement by specialty engineering domain experts (e.g. safety, reliability) as well. For example, to address the impacts of system quality requirements on software, software engineering experts need to coordinate design decisions not only with systems engineering experts, but also with specialty engineering experts. The examples illustrate the kinds of "close coupling" of interrelated design decisions that can exist and thread across the system (hardware, software, and specialty engineering domains). When any of the major disciplines have insufficient influence during early system design activities, interrelated design dependencies are often overlooked. Discovered later on in the life cycle, addressing these dependencies becomes a major contributor to, at best, cost and schedule overruns, at worst, delivery of reduced capability or a cancelled program.

First Example – Project A

During development of a major satellite system (which was ultimately cancelled), hardware was the focus during early system engineering and architectural design activities. This hardware-centric approach resulted in an integrated master schedule (IMS) in which software development tasks would not even be planned until the hardware design was completed.

This is an extreme example of a project team following a system development method that isolates and delays engagement of software domain experts. We observed this type of approach as a significant source of high risk to cost and schedule overruns. The program's ability to meet technical requirements, many of which relied on software to deliver functionality, was put in jeopardy as well. In our judgment, had the program not been cancelled, the delayed planning of software development activities would likely have uncovered insufficient schedule time and staff remaining to meet key project delivery milestones.

Second Example – Project B

In this example, a software-reliant system under development includes heavy concerns for safety. That is, the system includes certain "threads" of functional operation provided mainly by software that, if failures occur, may result in serious human injury or even death.

Such systems typically incorporate regular "health and status monitoring" (HSM) messages generated by various constituent software components throughout the system. These HSM messages are proactively analyzed by software specifically designed to report on a variety of potential and/or detected system malfunctions. The intent is to identify problems in the system sufficiently early so as to minimize the risk of harm to humans.

In this project, the SEIT's functional decomposition efforts resulted in system functionality allocated to hardware and software configuration items which included a configuration item specifically for HSM functions. These configuration items were assigned to various IPTs before the IPTs themselves were fully formed and engaged in system development. In this large program, well over a year of time had elapsed before software engineers, segregated and with time-staggered starts across several product development IPTs, began independently designing and implementing the details of HSM message formats and contents.

Nearly two years after various product development IPTs started their designs, it was discovered that the HSM message designs were evolving inconsistently. Safety domain experts found it difficult and time consuming to scrub through the content of all the various HSM messages and message formats that were being developed, to identify which specific data were to be deemed safety related. Safety certifying authorities, external to the program, had to be convinced to approve the safety aspects of the system's design. This task would be much harder with a system software design that lacked consistent identification and representation of safety related information. Thus, redesign of HSM messages that carried safety information had to be negotiated across IPTs, and software engineers had to re-implement such messages.

This project illustrates a difficulty resulting directly from isolating and delaying involvement of software engineering domain experts during early phases of system design. We believe that early involvement in systems design by software architects would have significantly improved chances to discover which key software information structures, such as messages used by the HSM configuration item, must be defined consistently across the system. This is an important part of what software architects do; i.e., considering driving systems quality attribute requirements such as safety, security, etc., software architects make the "big software design decisions" that should not be delegated to downstream developers in individual IPTs. We believe such an approach would have averted the schedule delays associated with the downstream negotiation and redesign of HSM message content across the various IPTs

In addition to early inclusion of software domain experts, we believe a further reduction in schedule risk could have been realized by early inclusion of specialty engineering experts in this project's systems design activities. For example, had safety engineers been involved in the up-front system design process, they would have had earlier opportunities to identify safety related information within software HSM messages. This would have enabled safety engineers to have more time to build evidence for constructing safety cases to be made to certifying authorities.

Third Example – Project C

In this project, a software-reliant system was being developed that involved a control system and a physically separate vehicle system (e.g., a satellite, unmanned air vehicle system, remotely controlled surface vehicle, etc.).

With these types of systems, it is crucial to collect and display information that provides humans with "situational awareness" (SA) of the vehicle's state and the state of the surrounding environment. To achieve a harmonious assessment of a vehicle's SA, it is important that various humans interacting with the system understand the "common operating picture" (COP). Designing a software system with a smooth integration of SA/COP information requires that software engineers work closely with human factors (HF) specialty engineers.

In this case, once the individual software teams were formed within the IPTs, each IPT synthesized its own notion of SA/COP functionality and related data. When software engineers from each IPT subsequently began working with HF specialty engineers, the HF engineers discovered difficulties in combining the two distinct notions of SA/COP and related information into an integrated SA/COP display. This discovery occurred many months into the design cycle,

requiring significant schedule delays to negotiate a harmonized SA/COP redesign within each IPT.

System capabilities such as SA/COP that are largely implemented in software often exhibit cross-cutting system-wide attributes and behavior that emerge and become discernible only when the system is considered or integrated as a whole. These attributes and behaviors belong to the overall system and thus are not inherent or allocable properties of any "piece" of the system.

Each IPT within this project determined it was responsible for the implementation of the SA/COP resulting in a belief that there was no reason to coordinate with other IPTs. We believe that had the architecture significantly defined the SA/COP, the IPTs would have understood the system context of the requirements and would not have constructed incompatible designs.

Fourth Example[7] - Project D

In this section, we summarize an example of a commercial company whose primary business is the manufacture and worldwide sales of diesel engines. A quick summary is that this company, Cummins Engine, is the world's largest manufacturer of diesel engines of more than 50 horsepower.

The different variations of engines involved are staggering: they range in horsepower from 50 to above 3,500, have from four to 18 cylinders, and operate with a wide variety of fuel systems, air handling systems, and sensors. The engines operate all over the world, requiring different operator interfaces and communications/datalink capabilities, and must be serviceable by vastly differing distribution and service infrastructures. The difficulty of managing such a breadth of product variation levied additional complexities on Cummins' already complex software. To address the complexity, Cummins' management elected to leverage a software technology based on software product lines. Adoption of this technology is accompanied by significant, company-wide commitment levels involving culture, mindset and procedural changes. Among the key changes in this company include having software architects involved with business managers and other engineering experts during the initial conception of a new diesel engine model.

---

[7] Content for this example taken directly from Chapter 9 of *Software Product Lines: Practices and Patterns*. P. Clements and L. Northrop. Addison-Wesley, 2002.

When creating a new model of diesel engine, the earliest system architectural decisions include the full participation of software domain experts.

Cummins pursued a course of action regarding its approach to software development that is the opposite of treating software as just another specialty engineering domain. In fact, this company recognized, in order to survive, it must establish software as one of its key core business competencies. In the end, Cummins acknowledged that it had essentially become a software company that manufactures engines.

RECOMMENDATIONS

To help avert the types of difficulties described in the above examples, we compiled a list of recommended actions for acquisition programs of record to consider adopting. The recommendations are arranged in an order that, in the authors' opinions, will provide the greatest benefit when incorporated.

1. Employ a formal systems architecture team and associated process within the government acquisition organization, whether the organization is at the program level, center level, service level, or DoD/federal level. For example, the organizational team and associated processes may include coordination of architecture development across multiple acquisition programs.

2. When creating the program work breakdown structure (WBS), be cognizant that you are levying architectural decisions which can/will constrain potential contractors into an architectural structure. To mitigate the potential of encroaching on the dangerous territory of starting to design the system within the WBS, designate an architecture team (i.e., enterprise level or peer level) to review the WBS for acceptability of any constraining architectural decisions. If an architectural review is not possible, the associated risk of omitting this step must be acknowledged, accepted, and documented.

3. Do not delegate important, key system characteristics to be decided or negotiated by downstream programs and/or product teams. Create and document architectural principles, key architectural design decisions and associated rationale, guidance and constraints for subsequent consumption by downstream programs and product teams. This will help to avert the costs and delays often incurred when addressing late discovery of key architectural drivers after significant design work has been done.

4. Physically and psychologically segregate the hardware from systems engineering. To help achieve this:

   a. Appoint a dedicated lead systems engineer

   b. Appoint a dedicated lead hardware engineer

   c. Appoint a dedicated lead software engineer

   Each of these engineering domains has distinct concerns, unique knowledge and practices. A lead engineer, as the primary point of contact for each domain, will help ensure each domain receives appropriate focus.

5. Organize software and hardware disciplines to have the same reporting chains as the systems engineering discipline.

6. Staff the architecture and software disciplines within the acquisition organization to accurately reflect the necessary skill levels, amount of work, and domain complexity to support the system. Ensure ongoing architecture and software training is implemented.

7. Ensure software and specialty engineering disciplines are key members of the architecture teams.

8. Increase early emphasis on iterative system architecture and design activities that include domain representation from software architecture engineering, specialty engineering and requirements engineering.

9. Have the architecture team along with the systems engineering discipline take a larger role in oversight of the IPTs' design efforts. Doing so implements the systems architecture function as part of systems engineering activities. The architecture team should establish the constraints that need to be followed by all downstream IPTs.

10. Avoid segregating software and traditional specialty engineering effort from the initial system architecture for purposes of reducing the initial cost estimates. This approach often results in significant redesign work with associated cost and schedule overruns.

11. Ensure strong architectural discipline from enterprise level down to individual IPTs.

12. Consider an iterative development approach (i.e., build a little – integrate a little – test a little). Small, iterative builds ensure an early integration of

software that reduces life-cycle software development and integration risk. Caveat: investigate how your specific program milestones are suited for agile-like development methods.

13. Ensure architecture is designed to promote future evolvability for integrating new software technologies as well as for integrating new capabilities long past initial system deployment. Designing a system that can be readily evolved does not happen by accident. Software systems that provide significant functionality typically need to endure within the operating environment many years after their initial deployment. At a minimum, an architecture which enables replacement of obsolete software technologies without causing major disruption to other parts of the system is crucial to constraining a system's full life-cycle costs. If such evolvable qualities are not considered in the early phases of system design, large risks to full life-cycle software costs will likely materialize as issues.

SUMMARY

Today's complex systems, even with the most cutting-edge hardware, would not properly function without successful and timely integration of software. As illustrated in the real program examples above, expertise from the software and specialty engineering disciplines may be delayed or completely overlooked during the initial systems engineering and architecture design activities. This behavior often results in systems that are late to enter system integration activities and/or which are over budget. These systems, when initially deployed, typically operate below expectations. Indeed, such systems may not meet their basic design requirements while exceeding projected development costs.

In today's environment of shrinking DOD budgets and cancelled programs, software upgrades are being used more and more to keep systems viable and relevant much longer than the systems' original creators anticipated (sometimes on the order of decades). It is thus essential that the upfront system and software architecture be "right" to promote the ease of frequent enhancement and support over the long term. Since the majority of a software-reliant system's full life-cycle expense is for creating and sustaining software,[8,9] proper up-front design/architecture significantly affects the system's total cost of ownership.

We recommend that any organization acquiring software-reliant systems should strive to incorporate software, as well as traditional specialty engineering disciplines, as equal partners from the initial system design and architecture through fielding. It is imperative that software and specialty engineering domains do not miss their opportunity to have important positive influence on early system architecture design decisions.

---

[8] D. Galin. *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison-Wesley, 2004.

[9] U.S. Air Force Space and Missiles Systems Center (SMC). *SMC Systems Engineering Primer & Handbook: Concepts, Processes and Techniques, 3rd Edition*, April 2005 (page 147).

ACKNOWLEDGEMENTS

The authors express their appreciation for the valuable inputs, comments and careful reading of draft versions of this document. Their insights have significantly improved content and readability:

Greg D. Blank, Defense Contracts Management Agency

Peter Capell, Software Engineering Institute

Julie Cohen, Software Engineering Institute

John Foreman, Software Engineering Institute

Theodore Marz, Software Engineering Institute

John Robert, Software Engineering Institute

Jeffrey Thieret, Software Engineering Institute

Eileen Wrubel, Software Engineering Institute

APPENDIX

The following excerpt of "Issue 2" is taken from *Top Software Engineering Issues within Department of Defense and Defense Industry*, *NDIA Task Group Report*, September 2006. While this reference is slightly dated, we are still observing this exact issue in presently executing programs.

Issue 2:

Fundamental system engineering decisions are made without full participation of software engineering.

The following main points provide amplification of this issue:

- Complex, distributed, interoperating systems and evolving software capabilities have permanently altered the system level trade space. Key architectural decisions early in the system life cycle have great impact on software capabilities, attributes, and architectural/design approaches, yet the software engineering discipline is not consistently involved in these decisions.

- Software engineering involves systems thinking as much as it does technology. Software engineers need the knowledge, skills, and authority to fully participate in system-level decision-making from program onset; even conceptual trades require software expertise.

- System engineering and software engineering life cycles, processes, and products are not always consistent or sufficiently harmonized for meaningful cross-discipline participation to occur.

- Proposal guidelines can impede cross-discipline cooperation and coordination by segregating SW and system activities and documents.

- In the planning phase, system development methods do not properly leverage SW's ability to rapidly field enhanced capabilities, a key need in the evolving acquisition environment. Yet many programs end up relying upon this ability during development and support phases.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0000749