SYM-AM-18-057

PROCEEDINGS
OF THE
FIFTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

WEDNESDAY SESSIONS
VOLUME I

Acquisition Research:
Creating Synergy for Informed Change

May 9–10, 2018

Published April 30, 2018

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Software Is Consuming DoD Total Ownership Cost

Brad R. Naegle, LTC, U.S. Army (Ret.)—is a Senior Lecturer within the Graduate School of Business and Public Policy at the Naval Postgraduate School (NPS). He has served as the Academic Associate for Program Management Curricula at NPS and has managed Distance Learning graduate degree and certificate programs. He currently serves on the Navy's software community of practice. On active duty, he was the PM, 2½ Ton Extended Service Program and the Deputy PM for Light Tactical Vehicles. He holds a master's degree in Systems Acquisition Management from NPS and a BS from Weber State University in economics. He is a graduate of the Command and General Staff College and Combined Arms and Services Staff School. [bnaegle@nps.edu]

Abstract

Department of Defense (DoD) software-intensive systems and the software content in other systems will continue to grow and may dominate total ownership costs (TOC) in the future. These costs are exacerbated by the fact that, in addition to contracted development costs, the bulk of software sustainment costs are also contracted. All of these factors indicate that DoD system software will continue to be a very expensive portion of TOC.

The software engineering environment remains immature, with few, if any, industry-wide standards for software development or sustainment. The Defense Acquisition System (DAS) is significantly dependent on mature engineering.

System software size and complexity are key indicators of both development costs and sustainment costs, so initial estimates are critical for predicting and controlling TOC. Unfortunately, the software size estimating processes require a significant amount of detailed understanding of the requirements and design that is typically not available when operating the DAS without supplementary analyses, tools, and techniques. Available parametric estimating tools require much of the same detailed information and are still too inaccurate to be relied upon. Similarly, understanding the potential software complexity requires in-depth understanding of the requirements and architectural design.

It is clear that the DoD must conduct much more thorough requirements analyses, provide significantly more detailed operational context, and drive the software architectural design well beyond the work breakdown structure (WBS) functional design typically provided. To accomplish this, the DAS must be supplemented with tools, techniques, and analyses that are currently not present.

System Software Development and Sustainment Environmental Challenges

While many of the TOC initiatives apply equally to hardware-oriented systems and software-oriented systems, there are some significant differences in both the software development and sustainment environments that need to be considered to gain better software-TOC performance. Understanding these differences in environments will help managers at all levels better manage the acquisition management system and provide the warfighter with systems that are easier and cheaper to sustain.

The Software Engineering Environment (Naegle, 2015)

The software engineering environment is not mature, especially when compared to hardware-centric engineering environments. Dr. Philippe Kruchten (2005) of the University of British Columbia remarks, "We haven't found the fundamental laws of software that would play the role that the fundamental laws of physics play for other engineering disciplines" (p. 17). Software engineering is significantly unbounded because there are no physical laws that help define environments. There is significant evidence for software engineering

immaturity, and it is nearly impossible to find widely accepted, industry-wide development standards, protocols, architectures, or formats. There is no dominant programming language, design and development process, standard architectures, or software engineering tools, which means that reusable modules and components rapidly become obsolete. All of these combine to make it nearly impossible to institute a widely accepted software reuse repository. Without significant software architecture and code reuse in developing software-intensive weapon systems, each development process essentially starts from scratch. This fact is one of the main reasons that the Technology Readiness Assessment (TRA) and the software Technology Readiness Levels (TRLs) are ineffective in predicting software development risk (Naegle & Petross, 2007).

The software engineering state-of-the-practice currently is wholly dependent on the requirements and operational environment cues that are passed to the software development team. From the requirements, a software architecture is designed, and the requirements "flow down" through that architecture to the individual modules and computer software units that are to be constructed. The software build focuses on the requirements that flowed down to that level and the integration required for functionality. The standards, protocols, formats, languages, and tools used for the build will likely be unique to the contractor developing the software, and will most certainly not be universally accepted or recognized across the software industry.

The software architectural design is the basis for all of the current and future system performance, including TOC performance, that the system will achieve, and the current state-of-the-practice in software engineering has each project design a unique architecture. Like hardware, the software design will significantly impact system attributes that are important to the warfighter, including TOC-oriented elements of maintainability, upgradability, interoperability, reliability, safety, and security. Most hardware-oriented engineering environments address these critical areas through widely accepted industry standards. For example, all DoD ground combat vehicles use a 24 volt, direct current, negative ground electrical system. Any current or future subsystem requiring vehicle power will automatically be designed to operate using those industry-wide electrical power standards.

The software engineering environment is in stark contrast to even our most advanced hardware-centric engineering environments. For example, in the automotive engineering field, a design that provides for easy replacement of wear-out items such as tires, filters, belts, and batteries obviously provides sustainability performance that is absolutely required. This engineering maturity helps account for derived and implied requirements not explicitly stated in the performance specification. Most performance specifications do not explicitly address this capability because they would be automatically considered by any competent provider within the mature automotive engineering environment. A mature engineering environment includes design elements and industry-wide standards, processes, materials, and techniques to which we have grown to expect. A significant problem will exist if we expect the software engineering environment to perform the same way as other, more mature engineering fields (Naegle & Petross, 2007).

As the example above illustrates, many system TOC elements are often standardized across hardware-oriented engineering environments due to the maturity of the sector's engineering maturity. Without the engineering maturity, software sustainability performance and expectations must be specified as part of the requirements generation process. The capabilities-based user requirements and performance-based acquisition requirements are specifically not designed to provide that level of specificity.

The Software Engineering Environment Challenge

The DoD's acquisition management system is designed to garner innovation from the commercial marketplace by leveraging the mature engineering environments present in most disciplines. The DoD develops its requirements beginning with the capabilities-based language provided by the users, then translating them into performance-based language for the RFP. This requirements generation system is purposely designed to allow the maximum contractor flexibility in satisfying the warfighter's needs.

Within the immature software engineering environment, this requirements generation process creates an opportunity for significant misinterpretation, and derived and implied requirements that are not addressed, all resulting in requirements creep that fuels cost increases and schedule slippage. Unlike mature hardware-oriented engineering environments, where the widely accepted industry standards will be employed whether or not they are specified, with software, you get what you specify and very little else (Naegle, 2015, p. 13).

Addressing the Challenge

There are several necessary steps to effectively address the immature software engineering environment challenge:

1. The acquisition community must understand that the software engineering environment is different, and not mature. This must be an essential part of Knowledge Point 1 and of the Navy gate reviews 1 through 5, detailed earlier. The BBP memoranda help support this step by its direction to "improve the professionalism of the total acquisition workforce."
2. The acquisition community must take active steps to compensate for the software immature engineering environment.
    a. Requirements. Fully develop all requirements so that derived and implied requirements are specified. Sustainment performance including maintainability, upgradability, interoperability, reliability, and safety/security must be specified to improve TOC attributes.
    b. Operational context. Provide context for the requirements beyond what is provided in the typical OMS/MP. Software engineers need to understand how the system will be used and maintained, how it will be modified and interfaced in the future, which features are critical and which are non-critical enhancers, and how the user expects the system to operate under stressful conditions at the limits of the operational envelope. All of this required information is not available from any other source, and certainly not available in the software engineering environment.
3. The acquisition community must drive and monitor the software architectural design process to a much greater extent than what is needed for hardware-centric system. This is an essential function to reach Knowledge Point 2, and you literally could not achieve Knowledge Point 2 without the ability to drive the software architectural design. This would also be an essential function to effectively pass through the Navy gate reviews 4 through 6.

Estimating Software Size and Cost

Estimating the software size is essential to estimating development and sustainment costs. Unfortunately, estimating size is difficult for any software-intensive effort, and nearly impossible for unprecedented development efforts, including many DoD weapon systems. The DoD often seeks cutting-edge technologies pursuing dominant capabilities, driving the need for developing unprecedented software development.

The Estimating Software Size and Cost Challenge

Estimating software size, especially for a cutting-edge weapon system, is challenging, at best. It is essential for understanding both software developmental and sustainment costs, so is critical to understanding TOC.

> Software Size Estimating is an important activity in software engineering that is used to estimate the size of an application or component in order to be able to implement other program management activities such as cost estimation or schedule progress. The software engineer is responsible for generating independent estimates of the software size throughout the life cycle. These estimates are sometimes expressed as Software Lines of Code (SLOC), Function Points (FP), or Equivalent Software Lines of Code (ESLOC). An effective software estimate provides the information needed to design a workable Software Development Plan (SDP). This estimate is also input to the Cost Analysis Requirements Description (CARD) process. ("Software Management," n.d., p. 1)

The U.S. Air Force has published a guide for weapon system software development management and describes the software estimating challenge as follows:

> Weapon system acquisition programs routinely aim to develop and deliver unprecedented warfighting capability. This unprecedented capability is often realized by developing complex, SIS [software intensive system] or integrating existing systems and subsystems with other equally complex systems in new ways. Since acquisition programs are planned and estimated when only top-level performance requirements are available, it is extremely difficult to develop high confidence estimates and align expectations early in the program life cycle. Such early estimates are relatively subjective, involve numerous assumptions, and are almost always optimistic since the engineering activities that result in a complete understanding of the work to be accomplished have not been completed. This complete understanding typically does not mature until well into the design phase, and when it does, it usually confirms that initial estimates were optimistic, key assumptions (such as significant reuse) cannot be achieved, more work than planned needs to be done, and the amount of software that has to be developed and/or integrated is growing. (SecAF, 2008, p. 7)

Both the AcqNotes website and the Air Force guidebook offer some guidance in estimating the amount of software that needs to be developed, which is not the only factor in the development cost, but certainly one of the most important.

The AcqNotes website recommends the following:

There are various ways available to the software engineer to develop a size estimate. It is recommended that multiple techniques be used and the results combined to produce the final size estimate. Methods that can be used of estimating size are:

- **Comparable to existing programs**: Compare the proposed functionality and other similarities to existing programs. If the proposed program has 20% more functionality than one program and 15% less than another, a fairly accurate estimate can be achieved using the actual sizes from the existing programs.
- **Historical data**: Within a program, historical data of previous developments (estimates and actual) may exist. Since many of the parameters are usually the same (developer team, environment, platform, etc.) this is a good method to compare previous software builds and the proposed code. The more data that is used will increase the accuracy.
- **Contractor estimate**: It is generally true the contractor has written software similar previously. They often maintain a database of past efforts (estimates and actual) and can produce a very accurate estimate. Since the contractor and the Government have different objectives, their estimate should never be relied on solely.
- **Expert judgment (Delphi technique)**: Engineers that have domain experience and knowledge can often accurately estimate the software size. Without extensive experience however, expert judgment is seldom more accurate than guessing.
- **Level of effort or schedule**: This method does not really estimate the size to be developed, but rather defines the most that could be developed given unchangeable level of effort or schedule constraints. The software engineer uses productivity rates, integration time and software defect data from recently delivered programs to define the maximum size that could be developed. ("Software Management," n.d., p. 1)

The Air Force guidebook also has recommended considerations for estimating software size:

> The software estimating process consists of a series of activities that include estimating size of the software to be developed, modified, or reused; applying estimating models and techniques; and analyzing, crosschecking, and reporting the results. The following steps should be considered as part of any software estimating process:
>
> - Develop a notional architecture for the system, and identify program requirements likely to be satisfied by software.
> - Identify potential COTS, GOTS, and other sources of NDI software.
> - Identify existing software that will be modified, including the size of the overall software as well as the size of the expected modifications.
> - Identify software that will be newly developed for this program to provide functionality not available from existing software, or to adapt/integrate all the necessary software components.
> - Obtain software size information for all software elements, where size is carefully defined and measured in one of the two standard software size measures: non-comment source lines of code (SLOC) or function points.
> - Assess the uncertainty in the new and modified software sizes, based on historical data (if available) and engineering judgment.

- Assess the uncertainty associated with the reusability of existing software (COTS, GOTS, and NDI) in the context of the program (see section 3.2.4). Estimate the trade studies, familiarization, and the integration and testing efforts required to accommodate the unmodified reused code.
- Account for software complexity and the proposed development approach/processes, and assess any overlaps in software builds.
- Be realistic about expected software productivity and any assumption of significantly higher than historical productivity due to applying the best people, improved/more efficient processes, or new and improved development tools. Past performance, where actual size, cost, and same program or a very analogous program, should be heavily weighted. It is rare to have the A-team people for a long-duration embedded system development, and new processes and tools often fall short of expectations.
- Apply growth factors to new/modified and reuse software, based on past experience and the level of uncertainty.
- Account for all remaining uncertainties as estimate risks (see section 3.2.2).
- Ensure the estimate includes software support to systems engineering, system and sub-system requirements definition, configuration management, quality assurance, program management, system integration, and system test as appropriate.
- Address the software development life-cycle from software requirements analysis through software-related system integration and testing. The chosen modeling/estimation approach may not address the entire software effort since some commercial parametric models focus on the period starting with the baseline set of software requirements and ending with a fully integrated and tested subsystem/functional software product ready for software/hardware integration and test. Estimate and include any additional effort required to develop, allocate, and analyze the subsystem and software requirements; perform software to hardware (subsystem) integration and test; and perform system integration and test.
- Crosscheck estimate results with other methods such as other models, expert advice, rules of thumb, and historical productivity.
- Improve the estimate over time. (SecAF, 2008, pp. 27–28)

Both the AcqNotes and U.S. Air Force size estimating guidance suggest using multiple methodologies to form a more informed estimate of the likely software size of a developmental system. Nearly all of the guidance is dependent on an excellent understanding of the system requirements and operational context.

One common method to estimate the software size on a new developmental program is to use the analogy method, that is, to compare the new system to a similar system that was recently developed, assuming that the software will be similar in overall size. The following is the first bullet in the AcqNotes software estimating guidance detailed previously in this section. It seems a logical approach, but has not proven particularly accurate in recent history:

> The premise is that the existing system's architecture, complexity, and functions are similar enough to fairly accurately predict the software development resources required for the new system. Unfortunately, this technique has proven to be ineffective as evidenced by the F-22 Raptor development and the follow-on F-35 Joint Strike Fighter (JSF) effort. The two high-performance, supersonic aircraft have overlapping missions, are significantly similar, and are both developed by the same contractor. The F-22 would seem to be a very good predictor of the F-35 software development effort with the SwTRL [Software Technology Readiness Level] model, but it clearly was not:
>
> The lines of code necessary for the JSF's capabilities have now grown to over 24 million—9.5 million on board the aircraft. By comparison, JSF has about 3 times more on-board software lines of code than the F-22A Raptor and 6 times more than the F/A-18 E/F Super Hornet. This has added work and increased the overall complexity of the effort. The software on-board the aircraft and needed for operations has grown 37 percent since the critical design review in 2005. … Almost half of the on-board software has yet to complete integration and test—typically the most challenging phase of software development. (GAO, 2012, p. 11)

The report goes on to state that typical software size growth in DoD systems development ranges from 30% to 100%.

> JSF design changes were originally supposed to taper off and be completed by January 2014. Actual design changes through September 2011 failed to taper off and continue at a significantly high rate. The projections in the GAO (2012) report indicated that the revised design change projections would continue and actually grow in number, until January 2019 (p. 16). Given this level of redesign, the software and system complexity growth are likely to continue. (Naegle, 2015)

The second bullet guidance from AcqNotes indicates that the use of historical data may be useful in estimating a new system's software size. This is particularly challenging for the DoD as the new weapon systems often have capabilities or features that are unprecedented (cutting-edge technologies). Certainly, there will be many subsystems in which historical data may be a good predictor for software size in existing, identical, or similar subsystems. However, the analogy method uses the historical data of a similar system as a surrogate for actual historical data, but suffers the challenges detailed previously.

The third AcqNotes bullet is "contractor estimates for software size." The problem with contractor estimates is that the size estimate is needed far before a development contractor would be involved in the process. Of course, market research contractors could be used to garner "contractor estimates," but this would require two essential preconditions. First, the market research contractor would need an extraordinary amount of requirements, operational context, and design detail on the proposed system to be able to provide to the marketplace to garner reasonably accurate software size estimates. Second, the market research would be conducted with industry members who can only respond to the information provided, so the estimates are only as accurate as the requirements-oriented information provided. In addition, the surveyed companies may be unwilling to provide much detail about their estimate as it could provide competitors with valuable competitive information.

The expert judgement, or Delphi Method (AcqNotes bullet 4), depends on the level of expertise of the engineers providing the estimate and their total understanding of the system to be developed. The DoD may gain access to expert software engineers that are inside the Government or through contracting for such expertise, but the level of understanding is dependent on the requirements generations system and the operational context provided.

There are also numerous parametric models, like Barry Boehm's Constructive Cost Model (COCOMO), that may be used in an attempt to estimate effort and cost (USC, 2002). COCOMO, like other estimating models, requires a software size estimate to be used. One of the inputs to the model is the Annual Change Traffic (ACT), or the percentage of the software that needs to be accessed for sustainability purposes. Obviously, the model would need to know the software size to perform the percentage calculations.

Because of all of the variables that are needed for the models, they can be quite misleading. For example, the University of Southern California (USC) used the models and then compared actual results to those estimated. They found that COCOMO "demonstrates an accuracy of within 20% of actuals 46% of the time for effort, and within 20% of actuals 48% of the time for a nonincremental development schedule" (USC, 2002). They found that, with more initial data input, the model accuracy improved to 30% of actuals 75% of the time. Boehm himself stated that "a software cost estimation model is doing well if it can estimate software development costs within 20% of the actual costs, 70% of the time, and on its home turf (that is, within the class of projects to which it is calibrated)" (SecAF, 2008, p. 21).

Obviously, using the results of parametric models alone would not result in the accurate estimates required by the DoD. The BBP memoranda specify "would cost" and "should cost" estimates that the models simply could not accurately produce. The software development cost and schedule estimate would necessarily need to be sufficiently accurate to avoid a Nunn-McCurdy violation in a software-intensive system development program.

Addressing the Challenge

Obviously, a fairly accurate software size estimate is necessary to predict both developmental and sustainment costs on a new system, and it is clear that obtaining an accurate size estimate is significantly challenging. The necessary precursor to software estimation is described earlier in this paper as compensating for the immature software engineering environment. Without more clearly defined requirements and operational context, accurately estimating software size is nearly impossible.

As suggested in both the AcqNotes and U.S. Air Force software estimating guidelines, a multi-faceted approach is needed. To be successful, each approach must be completed with significant discipline and rigorous systems analysis that goes beyond the current practices. If successful, the software size estimate will help predict both developmental and sustainment software costs.

Software Sustainability Architecture

A system's architecture and sustainability performance are strongly linked. Much of the design priority has been delegated to the contractor as the requirements language is capabilities-based on the user side and performance-based on the program management side. The DoD is responsible for driving the architectural design through the performance-based specification language, which requires a very in-depth understanding and development of the requirements.

The Software Architecture Challenge

Driving the software architectural design towards improved system TOC performance has numerous and complex challenges. The DoD requirements generation process is designed around the premise that the commercial marketplace has solutions for achieving the system performance specified by the DoD. This philosophy came from the acquisition reforms of the ’90s, when systems were much more hardware-oriented, and the associated engineering environments were mature. As the DoD has moved to software-oriented systems, the philosophy did not change, even though the software engineering environment is not mature. This has created a significant mismatch in what the DoD communicates and what it expects to be delivered. Much of the mismatch can be linked to the software engineering immaturity:

> The lack of software engineering maturity impacts both requirements development and design of the architecture. To compensate for the relative immaturity of the software engineering environment, the DOD must conduct significantly more in-depth requirements analysis and provide potential software developers detailed performance specifications in all areas of software performance and sustainability. This is a significantly different mind-set than the hardware-dominated systems acquisition of the past.
>
> In addition to the performance requirements, software architectures must be similarly shaped to include system attributes expected by the warfighter. Many DOD user representatives and acquisition professionals have grown accustom to the engineering maturity levels offered by the hardware-oriented systems that dominated past acquisitions. Providing the system requirements in the same fashion may not drive the architecture for needed attributes. As demonstrated by the F-35 JSF redesign problems, changing software architectures during the development cycle will likely be costly in terms of schedule and funding. (Naegle, 2014, p. 14)

The DoD also provides the top levels of the work breakdown structure (WBS) to provide cues to the necessary design structures, but like the requirements generation process, the communication through the WBS is often too vague or lacking in necessary detail for the software engineers to understand important aspects of the design.

> Contracts resulting from proposals that are based on underdeveloped, vague, or missing requirements typically result in catastrophic cost and schedule growth as the true demands of the software development effort are discovered only after contract award. (Naegle, 2015, p. 8)

The design metrics are very important to ensure that the software architecture is meeting the warfighter needs and expectations for the new system, including the TOC performance. Too often, this process serves to identify missing requirements or clarify vague requirements, causing significant requirements creep impacting the cost and schedule.

Addressing the Challenge

The requirements generation process, the Operational Mode Summary/Mission Profile (OMS/MP), the WBS, and the resulting performance specification and Government-specified functional architecture (top levels of the WBS) must drive the software engineer to develop the detailed system architecture to the total needs of the warfighter. The software engineering environment will not compensate for vague or missing requirements, and there are virtually no industry-wide standards for sustainability.

Processes to both drive the software architecture and monitor the design activities are unlike the contractor's hardware architecture activities and significantly more critical. Fifty percent or more of the software effort is expended in requirements and architectural design, which is far greater than typical hardware-oriented systems. This means that half or more of the software development resources have been used by the Preliminary Design Review (PDR), which occurs quite early in the developmental process. Requirements creep and software changes after the PDR are significantly disruptive to the design process and are costly in both funding and schedule. In addition, changes occurring after the design is complete are typically accommodated through the use of software patches. While these patches may function adequately, they typically weaken the software structure and add difficulty to the sustainment effort as they add lines of code, are not generally well documented, and add complexity to problem analyses in the deployed system.

Software Sustainment Activities

The Post Deployment Software Support (PDSS) structure—maintainers, software engineering tools, documentation, licenses, and so forth—must all be funded and in place at the initial deployment as software maintenance will likely be required immediately due to the complexity. Most of the DoD software sustainment effort is accomplished through Contracted Logistics Support (CLS) strategies, so the support contracts are critical to system deployment.

As with hardware-oriented systems, the software sustainability performance is significantly defined by the system architecture. The software engineering immaturity means that there are no industry-wide standards for software sustainability, so the DoD must drive the desired sustainability performance into the software design.

The two major components that help determine a system's software sustainment cost are software size (SLOC count) and complexity. Many of the effort estimating tools need the software size to estimate the number of software maintainers that need to be dedicated to the sustainment effort. Complexity factors are then added into the calculations.

The Software Sustainment Challenge

The DoD system acquisition process is driven through the performance-based specifications, program WBS functional architectural cues, and high-level OMS/MP and, therefore, relies heavily on the contractor's expertise backed by the industry's mature engineering environments. This process is not adequate for driving the software architecture to a sustainable design as the immature software engineering environment has no industry-wide standards for sustainability, so software sustainability performance must be totally driven through the DoD front-end processes.

Unlike even the most sophisticated hardware system, the software maintainers must have the same skill sets as the design engineers, and so the DoD is typically contracting for software engineers to maintain the software. The software sustainment cost factors include maintainers, software tools, license fees, and associated contract costs for most DoD systems. While the non-personnel costs can be considerable, the cost of the maintainers is usually the largest part of the sustainment cost because the DoD is typically contracting for software engineers to maintain the software components.

The events driving the need for software maintenance are not always within the control of the system's PM. As the DoD continues to network platforms into Systems of Systems (SoSs), each platform is subject to the network's complexities and interoperability requirements.

Addressing the Challenge

The solutions for addressing the software sustainability challenge are rooted in solving the other issues presented in this section, as they all tend to build on one another. The DoD needs to recognize that the software engineering environment is immature, significantly different than the hardware-oriented engineering environments. That immaturity renders much of the DoD front-end processes ineffective for software-intensive systems, so active steps augmenting the standard acquisition processes must be taken to compensate.

TOC performance is being influenced by the ever increasing software functionality of DoD systems, so improving TOC performance means effectively addressing software development and sustainability costs. The software costs and performance are dependent on how effective the acquisition front-end processes address them, and the standard DoD acquisition management system appears to be insufficient for the software components.

The software TOC issues presented, and their underlying causes, call for supplementary Systems Engineering Process (SEP) tools, techniques, and analyses to be applied to the DoD acquisition process. The following sections describe recommended tools, techniques, and analyses that would help address the issues presented. All of these are designed to work within the Defense Acquisition System (DAS).

Driving the Software Requirements and Architectures for System Supportability

While the tools and techniques described in this section were designed for the software components, they would be just as effective for any non-software component because they are Systems Engineering (SE) oriented. The SEP focus used does not attempt to separate software from other components, so all system components would benefit from using these tools and techniques.

Software Supportability Analysis

As with hardware system components, software supportability attributes must be designed into the system architecture. Many hardware-oriented engineering fields are now quite mature, so that a number of supportability attributes would be automatically included in any competent design, even if they were not specified by the user community. For example, the state of maturity for the automotive engineering field means that, in any automotive-related program, there would be supportability designs allowing for routine maintenance of system filters, lubricants, tires, brakes, batteries, and other normal wear-out items. There are few, if any, corresponding supportability design attributes that would be automatically included in even the best software construct. Virtually all of the software supportability attributes required must be explicitly specified because they would not likely be included in the design architecture without clearly stated requirements. With software, you get what you specify and very little else. So how does one ensure that required software supportability attributes are not overlooked?

Logistics Supportability Analysis (LSA), performed extremely early, is one of the keys for developing the system supportability attributes needed and expected by the warfighter. The F/A 18 Super Hornet aircraft was designed for higher reliability and improved ease of maintenance compared to its predecessors (“F/A 18,” n.d.) because of warfighter needs for generating combat power in the form of aircraft sorties available. The LSA performed on the F/A 18 determined that a design fostering higher reliability and faster maintenance turnaround time (the engines are attached to the airframe at 10 locations and can be changed in about 20 minutes by a four-man team) would result in more aircraft being available to the commander when needed. The concept for software LSA is no different, but

implementing sound supportability analyses on the software components has been spotty, at best, and completely lacking, at worst.

To assist in effective software LSA, a focus on these elements is key: Maintainability, Upgradeability, Interoperability/Interfaces, Reliability, and Safety & Security—MUIRS.

Maintainability

The amount of elapsed time between initial fielding and the first required software maintenance action can probably be measured in hours, not days. The effectiveness and efficiency of these required maintenance actions is dependent on several factors, but the software architecture that was developed from the performance specifications provided is critical. The DoD must influence the software architecture through the performance specification process to minimize the cost and time required to perform essential maintenance tasks.

Maintenance is one area in which software is fundamentally different from hardware. Software is one of the very few components in which we know that the fielded product has shortcomings, and we field it anyway. There are a number of reasons why this happens; for instance, there is typically not enough time, funding, or resources to find and correct every error, glitch, or bug, and not all of these are worth the effort of correcting. Knowing this, there must be a sound plan and resources immediately available to quickly correct those shortcomings that do surface during testing and especially those that arise during warfighting operations. Even when the system software is operating well, changes and upgrades in other interfaced hardware and software systems will drive some sort of software maintenance action to the system software. In other words, there will be a continuous need for software maintenance in the planned complex SoS architecture envisioned for net-centric warfare.

Because the frequency of required software maintenance actions is going to be much higher than in other systems, the cost to perform these tasks is likely to be higher as well. One of the reasons for this is that software is not maintained by ”maintainers,” as are most hardware systems, but is maintained by the same type of people that originally developed it—software engineers. These engineers will be needed immediately upon fielding, and a number will be needed throughout the lifespan of the system to perform maintenance, add capabilities, and upgrade the system. There are several models available to estimate the number of software engineers that will be needed for support; planning for funding these resources must begin very early in the process. Because the DoD has a very limited capability for supporting software internally, early software support is typically provided by the original developer and is included in the RFP and proposal for inclusion into the contract or as a follow-on Contractor Logistics Support (CLS) contract.

Upgradeability

A net-centric environment composed of numerous systems developed in an evolutionary acquisition model will create an environment of almost continuous change as each system upgrades its capabilities over time. System software will have to accommodate the changes and will have to, in turn, be upgraded to leverage the consistently added capabilities. The software architecture design will play a major role in how effective and efficient capabilities upgrades are implemented, so communicating the known, anticipated, and likely system upgrades will impact how the software developer designs the software for known and unknown upgrades.

Trying to anticipate upgrade requirements for long-lived systems is extremely challenging to materiel developers, but is well worth their effort. Unanticipated software

changes in the operational support phase cost 50 to 200 times the cost in early design, so any software designed to accommodate an upgrade that is never realized costs virtually nothing when compared to changing software later for a capability that could have been anticipated. For example, the Army Tactical Missile System (ATACMS) Unitary was a requirement to modify the missile from warhead air delivery to surface detonation—that is, flying the warhead to the ground. The contract award for the modification was $119 million. The warhead was not new technology, nor particularly challenging to integrate with the missile body. The vast majority of this cost was to reengineer the software to guide the missile to the surface. Had there been an upgrade requirement for this type of mission in the original performance specification, this original cost (including potential upgrades, even if there were 10 other upgrade requirements that were never applied) would have been a fraction of this modification cost.

Interfaces/Interoperability

OA design focuses on the strict control of interfaces to ensure the maximum flexibility in adding or changing system modules, whether they are hardware or software in nature. This presupposes that the system modules are known—which seems logical, as most hardware modules are well-defined and bounded by both physics and mature engineering standards. In sharp contrast to hardware, software modularity is not bounded by physics, and there are very few software industry standards for the modular architecture in software components. This is yet another area in which the software developer needs much more information about operational, maintenance, reliability, safety, and security performance requirements, as well as current, planned, and potential system upgrades. These requirements, once well-defined and clearly communicated, will drive the developer to design a software modular architecture supporting OA performance goals. For example, if a system uses a Global Positioning System (GPS) signal, it is likely that the GPS will change over the life of the system. Knowing this, the software developer creates a corresponding discrete software module that is much easier and less expensive to interface, change, and upgrade as the GPS system does so.

With the system software modular architecture developed, the focus returns to the interfaces between hardware and software modules, as well as to the external interfaces needed for the desired interoperability of the net-centric force. Software is, of course, one of the essential enablers for interoperability and provides a powerful tool for interfacing systems, including systems that were not designed to work together. Software performing the function of “middleware” allows legacy and other dissimilar systems to interoperate. Obviously, this interoperation provides a significant advantage, but it comes with a cost in the form of maintainability, resources, and system complexity. As software interfaces with other components and actually performs the interface function, controlling it and ensuring the interfaces provide the desired OA capability becomes a major software-management and software-discipline challenge.

One method being employed by the DoD attempts to control the critical interfaces through a set of parameters or protocols rather than through active management of the network and network environment. This method falls short on several levels. It fails to understand and control the effects of aggregating all of the systems in a net-centric scheme. For instance, each individual system may meet all protocols for bandwidth, but when all systems are engaged on the network, all bandwidth requirements are aggregated on the network—overloading the total bandwidth available for all systems.

> While these standards may present a step in the right direction, they are limited in the extent to which they facilitate interoperability. At best, they define a minimal infrastructure that consists of products and other standards

> on which systems can be based. They do not define the common message semantics, operational protocols, and system execution scenarios that are needed for interoperation. They should not be considered system architectures. For example, the C4ISR domain-specific information (within the JTA) identifies acceptable standards for fiber channels and radio transmission interfaces, but does not specify the common semantics of messages to be communicated between C4ISR systems, nor does it define an architecture for a specific C4ISR system or set of systems. (Morris et al., 2004, p. 38)

Clearly, understanding and controlling the interfaces is critical for effective interoperation at both the system and SoS levels. The individual PM must actively manage all systems' interfaces impacting OA performance, and a network PM must do the same for the critical network interfaces. Due to this necessity of constant management, a parameters-and-protocols approach to net-centric OA performance is unlikely to produce the capabilities and functionality expected by the warfighter.

Understanding the software interfaces begins with the software architecture; controlling the interfaces is a unique challenge encompassing the need to integrate legacy and dissimilar systems and the lack of software interface standards within the existing software engineering environment. As stated earlier, the architecture needs to be driven through detailed performance specifications, which will help define the interfaces to be controlled. An effective method for controlling the interfaces is to intensely manage a well-defined Interface Control Document (ICD), which should be a Contract Data Requirements List (CDRL) deliverable on any software-intensive or networked system.

Reliability

While the need for highly reliable weapon systems is obvious, the impact on total system reliability of integrating complex software components is not so obvious. Typically, as system complexity increases, maintaining system reliability becomes more of a challenge. Add the complexity of effectively networking an SoS (all of which are individually complex) to a critical warfighting capability that is constantly evolving over time, and reliability becomes daunting.

Once again, the software developer must have an understanding of reliability requirements before crafting the software architecture and developing the software applications. Highly reliable systems often require redundant capability, and this holds true for software components as well. In addition, software problems tend to propagate, resulting in a degradation of system reliability over time. For example, a Malaysian Airlines Boeing 777 suffered several flight control problems resulting in the following: a near-stall situation, contradicting instrument indications, false warnings, and difficulty controlling the aircraft in both autopilot and manual flight modes. The problems were traced to software in an air data inertial reference unit that was feeding erroneous data to the aircraft's primary flight computer (PFC), which is used in both autopilot and manual flight modes. The PFC continued to try to correct for the erroneous data received, adjusting flight control surfaces in all modes of flight, displaying indications that the aircraft was approaching stall speed and overspeed limits simultaneously, and causing wind shear alarms to sound close to landing (Dornheim, 2005, p. 46). It is critical for system reliability that the software developers understand how outputs from software applications are used by interfaced systems so that appropriate reliability safeguards can be engineered into the developed software.

Software that freezes or shuts down the system when an anomaly occurs is certainly not reliable nor acceptable for critical weapon systems, yet these characteristics are

prevalent in commercially based software systems. Mission reliability is a function of the aggregation of the system's subcomponent reliability, so every software subcomponent is contributing to or detracting from that reliability. The complexity of software makes understanding all failure modes nearly impossible, but there are many techniques that software developers can employ when designing the architecture and engineering the applications to improve the software component reliability. Once requirements are clearly communicated to the developers, the software can be engineered with redundancy or "safe mode" capabilities to vastly improve mission reliability when anomalies occur. The key is identifying the reliability requirements and making them clear to the software developers.

Safety & Security

Very few software applications have the required safety margins associated with critical weapon systems used by warfighters in combat situations—where they are depending on these margins for their survival. Typically, the software developers have only a vague idea of what their software is doing and how critical that function is to the warfighter employing the weapon system. Safety performance must be communicated to the software developers from the beginning of development so they understand the link between software functionality and systems safety. For example, suppose a smart munition senses that it does not have control of a critical directional component, and it calculates that it cannot hit the intended target. The next set of instructions the software provides to the malfunctioning system may well be critical to the safety of friendly troops, so software developers must have the necessary understanding of operational safety to decide how to code the software for what will happen next.

Software safety is clearly linked with reliability since software that is more reliable is inherently safer. It is critical that the software developer understands how the warfighter expects the software to operate in abnormal situations, in degraded modes, and when inputs are outside of expected values. Much commercially based software simply ceases to function under these conditions or gives error messages that supersede whatever function was being performed, none of which are acceptable in combat operations.

With software performing so many critical functions, there is little doubt that software applications are a prime target for anyone opposing U.S. and Allied forces. Critical weapon system and networking software must be resistant to hacking, spoofing, mimicking, and all other manners of attack. There must be capabilities for isolating attacks and portions of networks that have been compromised without losing the ability to continue operations in critical combat situations. The software developer must know that all of these capabilities are essential before he or she constructs software architectures and software programs, as this knowledge will be very influential for the software design and application development. The Software Engineering Institute's *Quality Attribute Workshop* states, "As an example, consider security. It is difficult, maybe even impossible, to add effective security to a system as an afterthought. Components as well as communication mechanisms and paths must be designed or selected early in the lifecycle to satisfy security requirements" (Barbacci et al., 2003, p. 2).

Interoperability challenges are increased when the SoS has the type of security requirements needed by the DoD. Legacy systems and existing security protocols will likely need to be considered before other security architecture can be effectively designed. OA capabilities will be hampered by the critical need for security; both must be carefully balanced to optimize system performance and security. This balance of OA and security must be managed by the DoD and not the software developer.

Physical security schemes and operating procedures will also have an impact on the software architecture. For example, many communication security (COMSEC) devices need only routine security until the keys, usually software programs, are applied; then, much more stringent security procedures are implemented. Knowledge of this security feature would be a key requirement of the developer; he or she must understand how and when the critical software pieces are uploaded to the COMSEC device. The same holds true for weapon systems that upload sensitive mission data just prior to launch.

Residual software on equipment or munitions that could fall into enemy hands presents another type of security challenge that needs to be addressed during the application development. For example, the ATACMS missile air-delivers some of its warheads, leaving the missile body to freefall to the surface. It is very conceivable that the body could be intact and, of course, unsecured. If critical mission software was still within the body and found by enemy forces, valuable information might be gleaned from knowing how the system finds its targets. The Government would certainly want the developer to design the applications in a way that would make anything recovered useless to the enemy, but this is a capability that is not intuitive to the software developers (Naegle, 2006, pp. 17–25).

Effective Software Development Tools Supporting System TOC Analyses

Software Engineering Institute's Quality Attribute Workshop

The Quality Attribute Workshop (QAW) is designed to help identify a complete (or as complete as possible) inventory of system software requirements through analysis of system quality attributes. One of the intents is to develop the derived and implied requirements from the user-stated requirements, which is a necessary step when user-stated requirements are provided in terms of capabilities needed as prescribed by the Joint Capabilities Integration Development System (JCIDS) process. A system's TOC, and those elements that contribute to TOC, are system quality attributes. Although obviously important to the warfighter, the associated operations and support, training/education, and facility costs are rarely addressed in much detail and need to be derived from stated requirements or augmented with implied requirements through the QAW process, or something similar.

The QAW helps provide a facilitating framework and process designed to more fully develop the derived and implied requirements that are critical to clearly communicate to potential contractors and software developers. Including a robust LSA process using the MUIRS focus elements, described previously, within the QAW process will likely significantly improve requirements analysis for those associated TOC elements and vastly improve the accuracy of system TOC projections. While improving the system requirements development, QAW is designed to work with another SEI process called the Architectural Tradeoff Analysis Methodology^{SM} (ATAM^{SM}) to further improve the understanding of the system for potential contractors and software developers.

SEI's Architectural Tradeoff Analysis Methodology^{SM}

The Software Engineering Institute's Architectural Tradeoff Analysis Methodology^{SM} (ATAM^{SM}) is an architectural analysis tool designed to evaluate design decisions based on the quality attribute requirements of the system being developed. The methodology is a process for determining whether the quality attributes, including TOC attributes, are achievable by the architecture as it has been conceived before enormous resources have been committed to that design. One of the main goals is to gain insight into how the quality attributes trade off against each other (Kazman, Klein, & Clements, 2000, p. 1).

Within the Systems Engineering Process (SEP), the ATAM[SM] provides the critical requirements loop process, tracing each requirement or quality attribute to corresponding functions reflected in the software architectural design. Whether ATAM[SM] or another analysis technique is used, this critical SEP process must be performed to ensure that functional- or object-oriented designs meet all stated, derived, and implied warfighter requirements. In complex systems development such as weapon systems, half or more than half of the total software development effort will be expended in the architectural design process. Therefore, the DoD PMs must ensure that the design is addressing requirements in context and that the resulting architecture has a high probability of producing the warfighters' JCIDS stated, derived, or implied requirements.

The ATAM[SM] focuses on quality attribute requirements, so it is critical to have precise characterizations for each. To characterize a quality attribute, the following questions must be answered:

- What are the stimuli to which the architecture must respond?
- What is the measurable or observable manifestation of the quality attribute by which its achievement is judged?
- What are the key architectural decisions that impact achieving the attribute requirement? (Kazman et al., 2000, p. 5)

The ATAM[SM] scenarios are a key to providing the necessary information to answer the first two questions, driving the software engineer to design the architecture to answer the third. This is a critical point at which all of the MUIRS elements need to be considered and appropriate scenarios developed.

The ATAM[SM] uses three types of scenarios: *Use-case scenarios* involve typical uses of the system to help understand quality attributes in the operational context; *growth scenarios* involve anticipated design requirements, including upgrades, added interfaces supporting SoS development, and other maturity needs; and *exploratory scenarios* involve extreme conditions and system stressors, including Failure Modes and Effects Criticality Analysis (FMECA) scenarios (Kazman et al., 2000, pp. 13–15). As depicted in Figure 1, the scenarios build on the basis provided in the JCIDS documents and requirements developed through the QAW process. These processes lend themselves to development in an Integrated Product Team (IPT) environment led by the user/combat developer and including all of the system's stakeholders. The IPT products will include a set of scenarios, prioritized by the needs of the warfighter for system capability. The prioritization process provides a basis for architecture trade-off analyses. When fully developed and prioritized, the scenarios provide a more complete understanding of requirements and quality attributes in context with the operation and support (including all of the MUIRS elements) of the system over its life cycle. A more complete understanding of the system's TOC elements should emerge from this type of analysis.

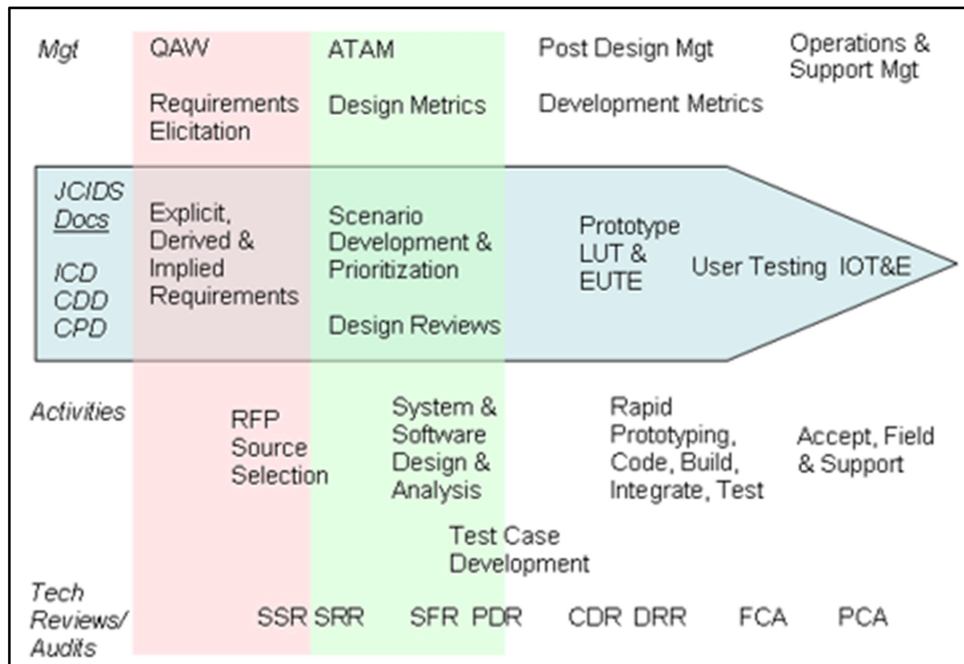

Figure 1. QAW & ATAM^{SM} Integration Into Software Life-Cycle Management

Just as the QAW process provides a methodology supporting RFP, source-selection activities, and the Software Specification and System Requirements Reviews (SSR and SRR), the ATAM^{SM} provides a methodology supporting design analyses, test program activities, and the System Functional and Preliminary Design Reviews (SFR and PDR). The QAW and ATAM^{SM} methodologies are probably not the only effective methods supporting software development efforts, but they fit particularly well with the DoD's goals, models, and SEP emphasis. The user/combat developer (blue arrow block in Figure 4) is kept actively involved throughout the development process—providing key insights the software developer needs to successfully develop warfighter capabilities in a sustainable design for long-term effectiveness and suitability. The system development activities are conducted with superior understanding and clarity, reducing scrap and rework, and saving cost and schedule. The technical reviews and audits (part of the DoD overarching SEP) are supported with methodologies that enhance both the visibility of the necessary development work as well as the progress toward completing it.

One of the main goals in analyzing the scenarios is to discover key architectural decision points that pose risks for meeting quality requirements. Sensitivity points are determined, such as real-time latency performance shortfalls in target tracking. Trade-off points are also examined so that TOC impacts resulting from proposed trade-offs can be analyzed. The Software Engineering Institute explained, "Trade-off points are the most critical decisions that one can make in an architecture, which is why we focus on them so carefully" (Kazman et al., 2000, p. 23).

The ATAM^{SM} provides an analysis methodology that complements and enhances many of the key DoD acquisition processes. It provides the requirements loop analysis in the SEP, extends the user/stakeholder JCIDS involvement through scenario development, provides informed architectural trade-off analyses, and vastly improves the software developer's understanding of the quality requirements in context. Architectural risk is significantly reduced, and the software architecture presented at the Preliminary Design

Review (PDR) is likely to have a much higher probability of meeting the warfighters' need for capability, including TOC elements.

Together, the QAW and ATAM^{SM} provide effective tools for addressing problem areas common in many DoD software-intensive system developments: missing or vaguely articulated performance requirements, significantly underestimated software development efforts (resulting in severely underestimated schedules and budgets), and poor communication between the software developer and the Government (both user and PM). Both tools provide frameworks for more detailed requirements development and more effective communication, but they are just tools—by themselves, they will not replace the need for sound planning, management techniques, and effort. Both QAW and ATAM^{SM} provide methodologies for executing SEP Requirements Analysis and Requirements Loop functions, effective architectural design transition from user to developer, and SEP design loop and verification loop functions within the test-case development.

A significant product resulting from the ATAM^{SM} is the development of test cases correlating to the use case, growth, and exploratory scenarios developed and prioritized. Figure 2 depicts the progression from user-stated capability requirements in the JCIDS documents to the ATAM^{SM} scenario development, and finally to the corresponding test cases developed. The linkage to the user requirements defined in the JCIDS documents is very strong as those documents drive the development of the three types of scenarios, and, in turn, the scenarios drive the development of the use cases. The prioritization of the scenarios from user-stated Key Performance Parameters (KPPs), Critical Operational Issues (COIs), and FMECA analysis flows to the test cases, helping to create a system test program designed to focus on effectiveness and suitability tests—culminating in the system Operational Test and Evaluation (OT&E). FMECA is one of the focus areas that will have a dynamic impact on TOC analysis because it will help identify software components that need higher reliability and back-up capability. The MUIRS focus helps ensure that TOC elements are addressed in design and test.

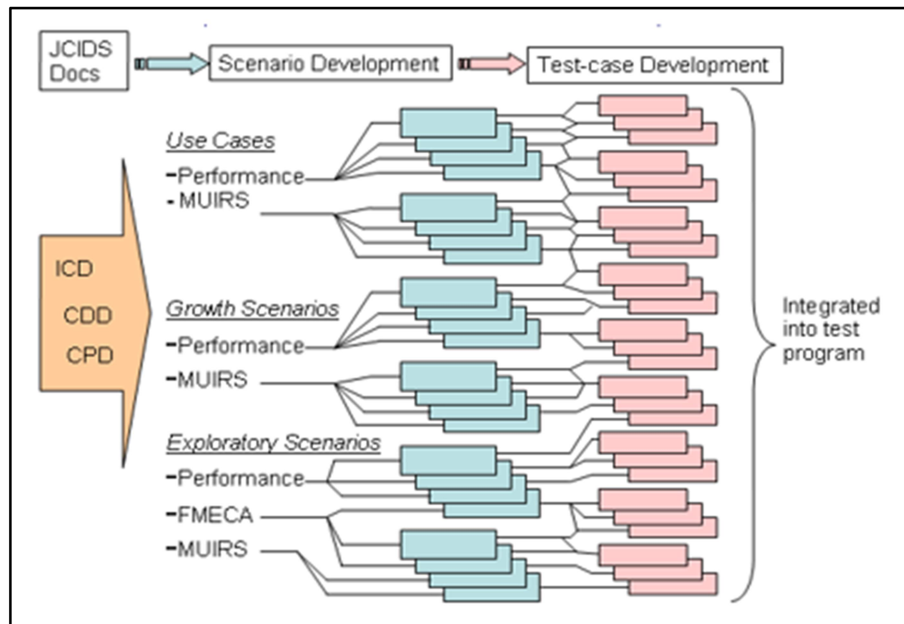

Figure 2. Progression From User-Stated Capability Requirements Through Scenario Development to Test-Case Development

Capabilities-Based ATAM^{SM} Scenario Development

The traceability from user-stated requirements through scenario development to test-case development provides a powerful communication and assessment methodology. The growth scenarios and resulting test cases are particularly suited for addressing and evaluating TOC design requirements because the system evolves over its life cycle, which is often overlooked in current system development efforts.

The software developer's understanding of the eventual performance required in order to be considered successful guides the design of the architecture and every step of the software development, coding, and testing through to the Full Operational Capability (FOC) delivery and OT&E. Coding and early testing of software units and configuration items is much more purposeful due to this level of understanding. The MUIRS and FMECA focus will help the design process for better TOC performance.

The resulting test program is very comprehensive as each prioritized scenario requires testing or other verification methodologies to demonstrate how the software performs in each related scenario and satisfies the quality attributes borne of the user requirements. The testing supports the SEP design loop by verifying that the software performs the functions allocated to it and, in aggregate, performs the verification loop process by demonstrating that the final product produces the capability identified in the user requirements through operational testing.

Both QAW and ATAM^{SM} require the capturing of essential data supporting decision-making and documenting decisions made. These databases would be best used in a collaborative IT system, as described in the next section.

Conclusions and Recommendations: Major Thrusts to Control Software Component TOC

Conclusions

DoD software-intensive systems and the software content in other systems will continue to grow and may dominate the TOC costs in the future. These costs are exacerbated by the fact that, in addition to contracted development costs, the bulk of the software sustainment costs are also contracted. In addition, the skill sets needed for software sustainment are the same as for software development, so the DoD is contracting for software engineers to perform maintenance functions. All of these factors indicate that DoD system software will continue to be a very expensive portion of TOC.

The software engineering environment remains immature, with few, if any, industry-wide standards for software development or sustainment. The Defense Acquisition System (DAS) is significantly dependent on mature engineering environments to compensate for the gaps and interpretation requirements presented with the performance-based specifications, vague Operational Mission Summary/Mission Profiles, and high-level work breakdown structures (WBSs) that the DoD provides during the request for proposal (RFP) process.

The developer software engineers will consume 50% or more of their contract resources analyzing requirements and developing the architectural design. This effort is expended before the Preliminary Design Review (PDR) and requirement additions (requirements creep), or changes beyond that point have disastrous effects on the software design and can even cause a complete redesign at extreme cost in funding and schedule.

The system software size and complexity are key indicators of both the development costs and the sustainment costs, so the initial estimates are critical for predicting and controlling TOC. Unfortunately, the software size estimating processes require a significant

amount of detailed understanding of the requirements and design that is typically not available when operating the DAS without supplementary analyses, tools, and techniques. Available parametric estimating tools require much of the same detailed information and are still too inaccurate to be relied upon. Similarly, understanding the potential software complexity requires in-depth understanding of the requirements and architectural design.

It is clear that the DoD must conduct much more thorough requirements analyses, provide significantly more detailed operational context, and drive the software architectural design well beyond the WBS functional design typically provided. To accomplish this, the DAS must be supplemented with tools, techniques, and analyses that are currently not present.

Recommendations

Program managers for software-intensive systems must supplement the DAS processes to

- compensate for the immature software engineering environment
- gain sufficient detailed information to perform reasonable software size and complexity estimates critical to understanding and managing system TOC
- complete the inventory of derived and implied requirements, including the often neglected sustainability requirements, before the RFP is issued
- provide more detailed system operational context, beyond what exists in most OMS/MP documents
- obtain more realistic contractor proposals in terms of cost and schedule associated with the software development and sustainment
- drive the software architecture for a more sustainable, less complex design
- monitor the software design process (metrics) to ensure the effort is progressing towards an effective, supportable, and testable design supporting the warfighter

The tools, techniques, and analyses presented in this research are designed to accomplish the tasks outlined above, and are compatible with the Systems Engineering Process (SEP) supporting the DAS. They also are designed to work together in a synergistic method to improve the software-intensive system development and sustainment performance influencing system TOC. They are certainly not the only tools, techniques, and analyses available to improve the process, and others may be as effective, as long as they can address the bulleted items above.

The maintainability, upgradability, interoperability, reliability, and safety/security (MUIRS) analysis technique is designed to help identify derived and implied requirements that need to be more fully articulated to ensure that the software engineer adequately considers these critical system attributes. These were selected because they are often missing from the user's capability-based requirements documents and the resulting performance specification, yet they are critical for the warfighter and are significant TOC drivers.

The Quality Attribute Workshop (QAW) is a technique to help more fully detail all requirements, including derived and implied. It is often used with the system WBS to more fully develop the desired functional design, especially when combined with the MUIRS analyses.

The Architectural Tradeoff Analysis Methodology^{SM} (ATAM^{SM}) is designed to be used with the QAW and provides detailed operational context through the scenario development,

providing critical design cues to the software development engineers. The scenarios include Use Cases (how the system will be used and maintained if fielded today), Growth Cases (how the system will likely change over its life cycle, including future networking), and Exploratory Scenarios (how the system is to operate under unusual or stressful conditions). This research recommends including the MUIRS analyses in the ATAM, as well as Failure Modes and Effects Criticality Analyses (FMECA) to identify critical functionality requirements.

Combined, the tools, techniques, and analyses provide a much improved understanding of the system and identify critical attributes that the software developers need to know to design an effective and supportable design. These tools help compensate for the immature software engineering environment, provide more detailed information needed to perform size and complexity estimates, and provide detailed operational context needed for proper software architectural design. They help produce superior RFPs and garner more realistic contractor proposals. They provide processes for monitoring critical software design activities and full test matrix crosswalks. All of these enhancements will help more accurately estimate and manage software TOC attributes.

References

Barbacci, M., Ellison, R., Lattanze, A., Stafford, J., Weinstock, C., & Wood, W. (2003, August). *Quality attribute workshops (QAWs)* (3rd ed.; CMU/SEI-2003-TR-016). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.

Dornheim, M. A. (2005, September). *A wild ride. Aviation Week & Space Technology, 163*, 46.

F/A 18. (n.d.). In *Wikipedia*. Retrieved from http://www.wikipedia.org/wiki/McDonnell_Douglas_F/A-18_Hornet

GAO. (2012, March 20). *Joint Strike Fighter: Restructuring added resources and reduced risk, but concurrency is still a major concern* (GAO-12-525T). Retrieved from http://www.gao.gov

Kazman, R., Klein, M., & Clements, P. (2000, August). *ATAM[SM]: Method for architecture evaluation* (CMU/SEI-2000-TR-004). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.

Kruchten, P. (2005, March/April). Software design in a postmodern era. *IEEE Software, 18*(2), 17.

Morris, E., Levine, L., Meyers, C., Place, P., & Plakosh, D. (2004, April). *System of systems interoperability (SOSI): Final report.* Pittsburg, PA: Carnegie Mellon University, Software Engineering Institute.

Naegle, B. R. (2006, September). *Developing software requirements supporting open architecture performance goals in critical DoD system-of-systems* (NPS-AM-06-035). Monterey, CA: Naval Postgraduate School.

Naegle, B. R. (2015, February 4). *Gaining control and predictability of software-intensive systems development and sustainment* (NPS-AM-14-194). Monterey, CA: Naval Postgraduate School.

Naegle, B. R., & Petross, D. (2007, October). *Developing software requirements supporting open architecture performance goals in critical DoD system-of-systems* (NPS-AM-07-104). Monterey, CA: Naval Postgraduate School.

Secretary of the Air Force (SecAF). (2008, August). *Weapon system software management guidebook.* Washington, DC: Author.

Software management: Software size estimate. (n.d.). In AcqNotes. Retrieved March 9, 2017, from http://www.acqnotes.com/acqnote/careerfields/software-size-estimate

University of Southern California. (2002, September). COCOMO. Retrieved March 9, 2017, from http://sunset.usc.edu/cse/pub/research/COCOMOII/cocomo_main.html

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net