Giant Slayer: Will You Let Software be David to Your Goliath System?

Stephen Blanchette, Jr.*
Software Engineering Institute, Arlington, Virginia 22203

DOI: 10.2514/1.I010448

Modern aerospace systems are marvels of engineering. Their amazing achievements are due in large part to software, yet relatively few project personnel, especially project management staff, understand or appreciate the significance of it. The lack of understanding and attention to software often dooms large aerospace projects to significant delay if not outright failure. As a motivation, several software-related aerospace incidents, from benign to calamitous, are recounted here to demonstrate the extent of the problem. Frequently repeated oversights witnessed on many large aerospace projects are then reviewed, with an eye toward what can (and should) be done to avoid the pitfalls. The intent is to empower the reader with information and ideas to prevent software from slaying the giant systems that it is built to support.

Nomenclature

t = time

I. Introduction

MODERN aerospace systems are marvels of engineering. Each generation sees new systems ply the air and the cosmos, traveling at speeds, in directions, and over distances previously unimaginable. Innovations like jet engines that can vector thrust, airliners that can carry many hundreds of passengers, inherently unstable airframes that can leave the ground at all, are incredible advances in aerospace system capability. But those ever-increasing capabilities come at a cost: a concomitant increase in scale and complexity. A great many of these amazing advancements are quite beyond the limits of human ability to control; they simply would not be possible were it not for increasing computational power. At the core of that processing power is software.

Figure 1 brings the point home starkly; the growth of software in aerospace systems has been quite impressive over the last 40–50 years, with some systems reaching as high as double-digit millions of source lines of code [1,2]. Further, as shown in Fig. 2, for U.S. military aircraft, more and more system capability is enabled by, or even delivered exclusively through, software [3]. Nor is it just functionality being driven by software; software also is a major driver behind total system cost. The cost of software for the Boeing 777 was about $800 million [4], and industry experts estimate that as much as 25% of the cost of a Boeing 787 goes toward software for guidance, navigation, and control systems.† The U.S. Department of Defense summarizes the situation well by saying, "Software is a key enabler for almost every system, making possible the achievement and sustainment of advanced warfighting capabilities. Development and sustainment of software is frequently the major portion of the total system life-cycle cost . . . " [5].

What is alarming, then, is that these giant systems, so very dependent on software for their success, often are developed by teams with comparatively little knowledge of software. The software staff generally represents a minority in the overall project teams. Worse, project management staff often has no software knowledge at all. It is rather typical to see software managed as an adjunct engineering function instead of as a first-class discipline. Aerospace education has not kept up with the trend. As a consequence, one of the most critical drivers of program success (or failure), the software, typically receives little oversight or attention until things begin to go very wrong. And then, of course, the genie is out of the bottle. In the end, the software, key enabler of large, complex aerospace systems, can slay the very giants it was designed to support.

II. Some Notable Events

Given the growing prominence of software in aerospace systems, it comes as no surprise that software sometimes is the root cause of a range of system problems.

The U.S. military increasingly finds itself bedeviled by software problems. The F-35 Lightning II, formerly known as the Joint Strike Fighter, repeatedly has found its way into the news due to software difficulties. One Government Accountability Office (GAO) report notes that the fighter jet's mission systems software has been subject to delays and functionality limitations that could cause overall program delays and limits to the plane's initial warfighting capability. The report goes on to state that the GAO was unable to determine what the initial capabilities of the aircraft might be due to the state of the software testing effort [6]. Indeed, the GAO's concerns proved justified. In May 2016, the F-35 again made headlines when the U.S. Department of Defense announced it would have to delay the initial operational test and evaluation of the aircraft from mid-2017 to calendar year 2018 due to problems with the software.

The U.S. Air Force's F-22 Raptor experienced serious trouble on its first overseas deployment in 2007. A flight of six Raptors was forced to abort a relocation sortie to Kadena Air Base in Japan after a software error related to crossing the International Date Line (the 180th meridian of longitude) caused the total failure of the navigation systems and a severe degradation of the communications systems. The planes were forced to follow their aerial refueling tankers back to Hawaii [7,8].

Received 31 December 2015; revision received 7 June 2016; accepted for publication 1 August 2016; published online 21 October 2016.

*Deputy Director, Client Technical Solutions, 4401 Wilson Boulevard, 10th Floor. Associate Fellow AIAA.

†Abrams, M., "Top 5 Aerospace Trends of Now and the Future," March 2013, https://www.asme.org/engineering-topics/articles/aerospace-defense/top-5-aerospace-trends-now-future [retrieved 31 May 2014].

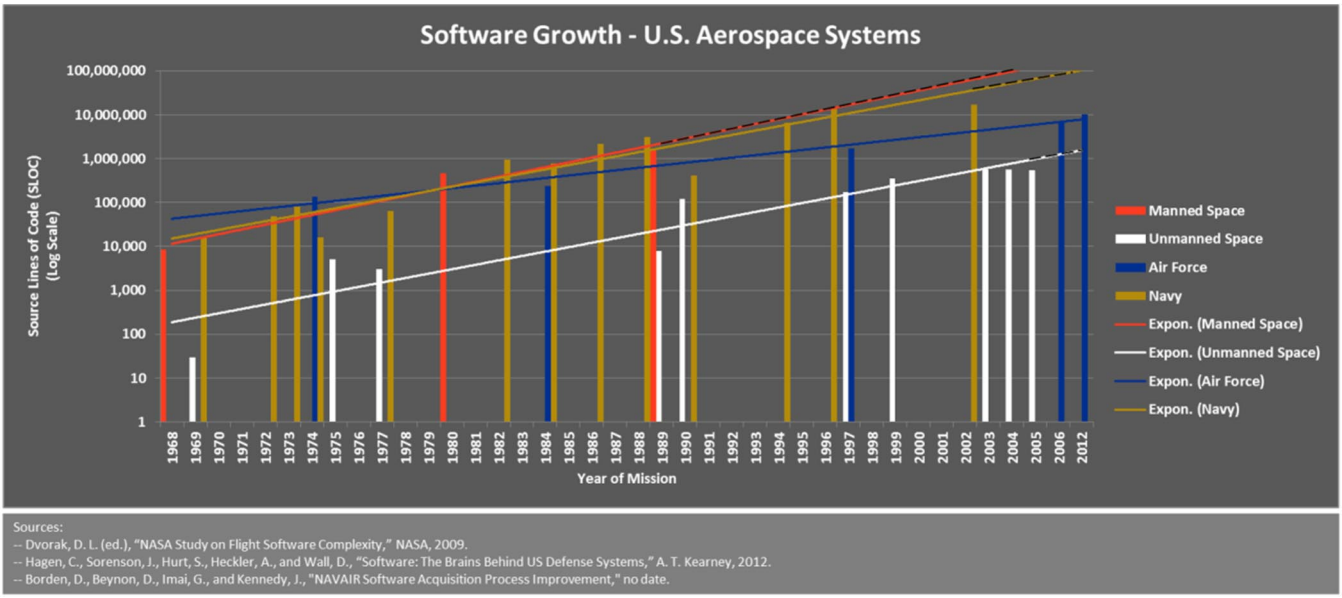


Fig. 1 Software growth in U.S. aerospace systems. Trendlines beyond the last data point (shown by dashes) are merely estimates based on extrapolation.

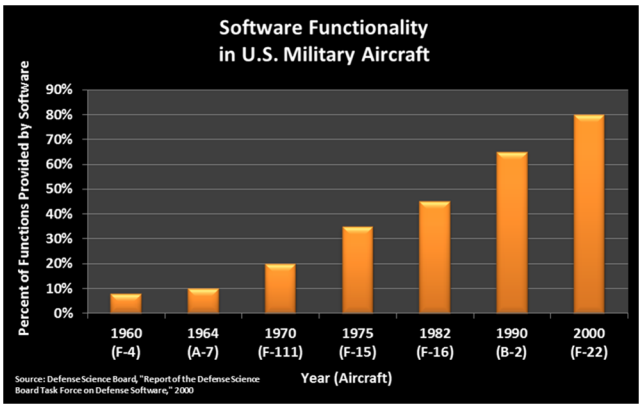


Fig. 2 Increasing functionality provided by software.

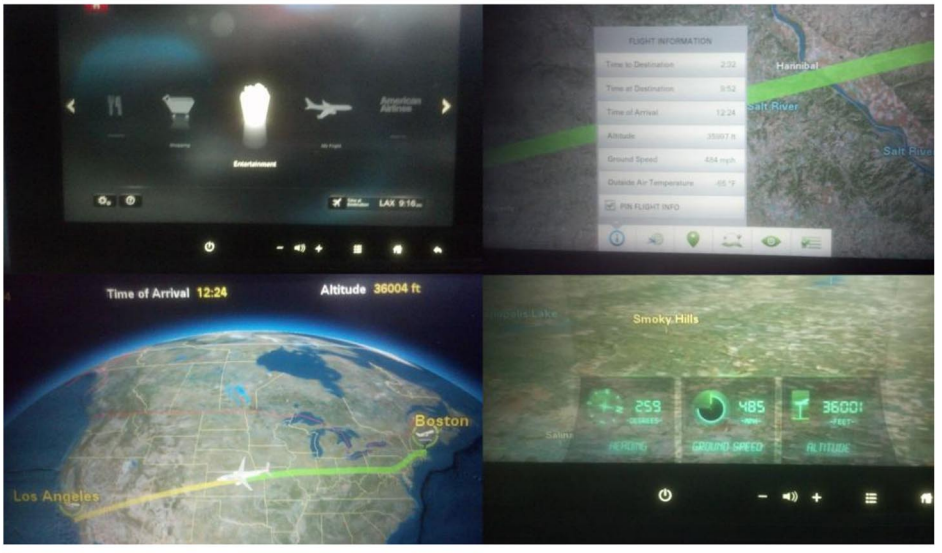
Commercial aviation, likewise, is experiencing its share of software challenges. Among the most serious is the case of Asiana Airlines flight 214, which crashed into the seawall at San Francisco International Airport while attempting to land in good weather in July 2013, resulting in three fatalities and over 100 injuries. The U.S. National Transportation Safety Board found that the "complexities of the autothrottle and autopilot flight director systems" of the Boeing 777 contributed to the controlled flight into stall accident and further recommended that Boeing redesign its wide-body automatic flight control systems [9–11]. For its part, Asiana Airlines also asserted that bad software design was a factor in the pilot's error.[‡]

In 2008, an Airbus A330 operated as Qantas Flight 72 experienced an "in-flight upset" that injured over 100 passengers, some seriously. In its investigation, the Australian Transport Safety Bureau found that the incident was "due to the combination of a design limitation in the flight control primary computer (FCPC) software of the Airbus A330/A340, and a failure mode affecting one of the aircraft's three air data inertial reference units (ADIRUs)." The rare combination of these two conditions caused the aircraft to pitch down suddenly (twice), causing many of the unrestrained passengers to be thrown to the ceiling [12].

In 2013 and 2014, the U.S. Federal Aviation Administration (FAA) issued airworthiness directives and special conditions related to software. The airworthiness directive addressed Boeing 747-8 and 747-8F series airplanes powered by certain General Electric engines, requiring the removal and replacement of "defective" electronic engine control software that the FAA determined could lead to "in-flight deployment of one or more [thrust reversers] due to loss of the . . . auto restow function, which could result in inadequate climb performance at an altitude insufficient for recovery, and consequent uncontrolled flight into terrain [13]." For the Boeing 787-8 type certification, the FAA added special conditions stipulating that the "design shall prevent all inadvertent or malicious changes to, and all adverse impacts upon, all systems, networks, hardware, software, and data in the Aircraft Control Domain and in the Airline Information Domain from all points within the Passenger Information and Entertainment Domain" [14]. A similar special conditions stipulation has been applied to the Boeing 777-200, -300, and -300ER series airplanes [15]. To appreciate the implications of the requirements levied in the 777 and 787 cases, consider Fig. 3, which depicts a theoretical example applicable to many modern airliners regardless of manufacturer. Among the many entertainment options now available to airline passengers at their seats is the ability to monitor the status of the flight, including aircraft position and speed, time, and even some meteorological information. Whereas just a few years ago, the information was simply made available (Fig. 3a) and one could choose to view it or not, recent advances now allow passengers to select which data they will see and how that data will be displayed (Fig. 3b). To make the data available, a passenger

[‡]Wald, M. L., "Airline Blames Bad Software in San Francisco Crash," March 2014, http://www.nytimes.com/2014/04/01/us/asiana-airlines-says-secondary-cause-of-san-francisco-crash-was-bad-software.html?_r=0 [retrieved December 2015].

a)



b)

Fig. 3 That's entertainment, and a potential vulnerability.

entertainment system must interface with the aircraft flight systems, and software must pass the data across the interface. That software interface potentially also provides an access point into critical aircraft systems from the entertainment system. The scenario discussed here is theoretical; claims that hackers have already taken control of airliners have yet to be substantiated. Nevertheless, interest in commercial airliner cybersecurity continues and extends beyond the FAA. In December 2015, U.S. Senator Edward Markey sent letters to each of the U.S. airlines as well as Boeing and Airbus, demanding details of how technologies such as inflight Wi-Fi are protected and requesting records for any cyber intrusion (successful or attempted) going back five years.[§]

The FAA, meanwhile, has had its own struggles with software. In 2012, the Office of the Inspector General at the U.S. Department of Transportation issued a report that found the FAA's En Route Automation Modernization (ERAM) program "experienced software problems that have impacted the system's ability to safely manage and separate aircraft." ERAM, a key enabler of the FAA's NextGen program, processes high-altitude air traffic flight information across the National Airspace System [16]. More recently, the FAA confirmed that an April 2014 incident that forced the temporary shutdown of the Los Angeles International Airport and affected air traffic around Southern California and Nevada involved a software error in the ERAM system.[¶]

One of the most well-documented aerospace failures attributable to software was the case of the maiden flight of the Ariane 5 rocket.[**] In June of 1996, Ariane Flight 501, less than a minute into its mission, veered off its flight path and self-destructed as a consequence. The investigating board found that the Ariane 5 Inertial Reference System (SRI) was substantially the same as that of the Ariane 4 but, due to subtle differences in requirements and large differences in actual trajectories between the two, the Ariane 5 SRI (both primary and backup) realized a software error that caused it to provide invalid data to the onboard computer, which executed the flight program and controlled the nozzles of the solid boosters and the Vulcain cryogenic engine [18].

[§]Leitschuh, M., "Myth vs. Reality: Can Hackers Really Control Airplanes?" *Security Magazine*, September 2015, http://www.securitymagazine.com/articles/86603-myth-vs-reality-can-hackers-really-control-airplanes [retrieved December 2015]. Bennett, C., "Dem Pressures Airlines for Cyber Defense Details," December 2015, http://thehill.com/policy/cybersecurity/261865-senator-pressures-airlines-for-cyber-defense-details [retrieved December 2015].

[¶]Moore, J., "Dragon Lady Stopped Traffic," May 2014, http://www.aopa.org/News-and-Video/All-News/2014/May/06/Dragon-Lady-stopped-traffic [retrieved December 2015].

**Dowson takes issue with the characterization of the Ariane 5 incident as a software failure, claiming it was a system engineering failure for allowing the correctly executing software to be reused in an inappropriate context [17]. Regardless, software, as part of the overall system, clearly factored into the accident.

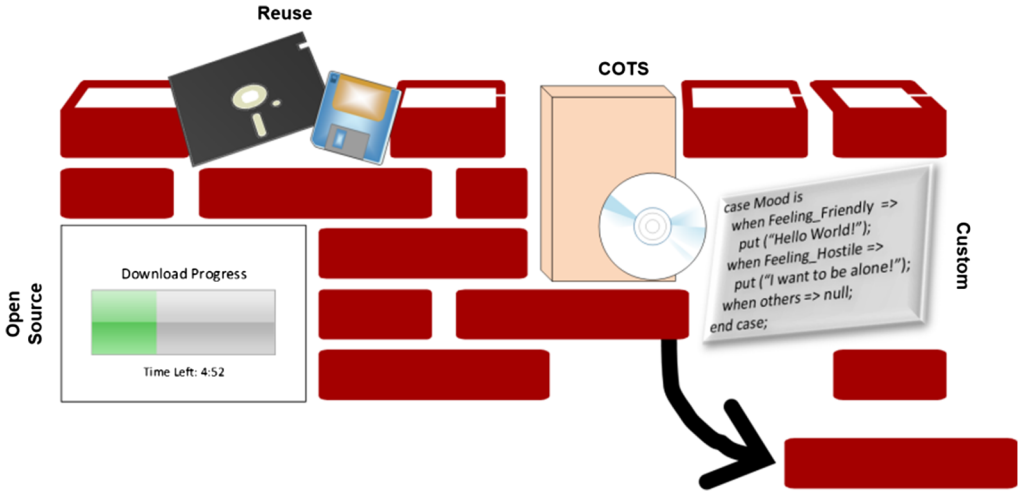


Fig. 4 A documented architecture allows reasoning about changes.

Perhaps the most shocking software-related aerospace failure is the case of the Airbus A400M military transport aircraft that crashed near Seville, Spain, in May 2015. Although investigation is ongoing as of this writing, the focus of investigators fell almost immediately, and has remained so resolutely, on the software involved in controlling the aircraft's turboprop engines. Four of the six crewmembers aboard died in the accident.

Many more such examples exist but the point has been made. Clearly, software can have an enormous effect on large, complex aerospace systems, and in more ways than often is imagined. Although the intent is that it does so in a system-enhancing manner, too often there are negative effects as well. Let us turn our attention to why this is so.

III. Typical Patterns of Project Behavior Leading to Problems

Considering the myriad opportunities for software to cause serious, even catastrophic, failures in complex aerospace systems, it is fortunate that actual occurrences have been relatively few. Yet the chances of such happenings persist, in part due to patterns that are repeated on program after program. Following are but a few of the common problems.

A. Missing the Details

Generally speaking, engineers, managers, and users alike often have a fairly good understanding of the basic requirements for large aerospace systems. The devil, of course, is in the details, and things often go wrong in three key areas.

The first is translating high-level system requirements into progressively lower-level requirements, all the way down to software. Undocumented assumptions and all manner of other communication issues often result in incomplete, erroneous, and even missing translations. Often, these errors are not discovered until testing begins, leaving a long trail of development artifacts to be corrected.

The second area is managing requirements. Invariably, circumstances will change during the course of development, causing perturbations in requirements. Many programs fail to plan for that eventuality, leading to unstable requirements (sometimes referred to as requirements "churn"), scope creep (resulting in an engineering effort that takes longer and costs more than originally planned), and deferred functionality due to the need to first address new or changed requirements.

The third area is accounting for the system quality attributes. Whereas regular requirements describe what a system must do, quality attributes describe how well the system must do what it does. Quality attributes are often called "ilities" because many of them end with the suffix "-ility" (e.g., scalability, availability, adaptability, etc.). However, quality attributes can include a broader perspective; performance, security, resilience, and so on also may be important quality attributes of large, complex aerospace systems. Often, quality attributes go completely unspecified; if they are documented, it is usually in vague and untestable ways (e.g., "the system shall be secure").

B. Failing to Architect the Entire System

It is hard to imagine developing a complex aerospace system without an extensive architectural design effort early on. Indeed, aerospace companies devote considerable resources toward architecting planforms, cable runs, hydraulic lines, avionics bays, and so on. Yet, having established that software undeniably is part of the system, it is astonishing how often a software architecture is not developed. One U.S. Department of Defense study noted that, based on in-depth review of 52 programs, inadequate software architectures is one of the top 10 systemic issues, whereas Croll asserts that the system architecture is incomplete without the software architecture [19].

The software architecture captures the structure of the software components, their properties, and the relationships among them [20]. A properly documented software architecture also provides the rationale for the architectural choices made during development. To be useful, a software architecture must be documented in multiple views, much like the architectural plan of a building consists of a structural view, an electrical view, a plumbing view, and so on [21]. In a complex system, no single engineer has complete cognizance of the entire software program (which itself often consists of many complex modules). In fact, the evolving nature of software development only exacerbates the problem. Increasingly, developers are turning to ready-made or readily available products[††] to fulfill key parts of the software requirements rather than crafting a totally bespoke solution. Updating or modifying the software in such a complex system is fraught with peril if developers do not have a documented software architecture to reference; the software architecture provides the basis for reasoning about the software. Figure 4 illustrates the point. All software has some inherent structure. Without knowing how the pieces fit together or why they are assembled in a particular way, one cannot determine, a priori, the consequences of removing or changing any given portion. In this example, removing the piece indicated by the arrow could introduce a security vulnerability or eliminate a vital element that the custom software relied upon for proper operation; absent the diagram (and supporting

[††]There are several types of ready-made and readily available software products including commercial off-the-shelf (COTS), government off-the-shelf, open-source software, and reuse of legacy software from previous programs.

rationale), such consequences are not obvious. Having a documented architecture allows one to make such assessments before committing to the change (and potentially incurring substantial rework).

C. Taking Too Narrow a View of Verification and Validation

Verification and validation (V&V) in large, complex aerospace systems tends to follow a remarkably simple pattern: model the pieces, model the whole, assemble the pieces into the whole, and then test the whole. This approach works fairly well for simple, physical things; it runs into problems in complex systems, whose very nature transcends the mere physical. Systems theory holds that the broad sweep of a system's characteristics cannot be understood simply from an understanding of the characteristics of individual constituent parts [22]. When one considers that aerospace systems operate in dynamic environments and interact with each other and with humans, this proposition becomes intuitive. Complex systems exhibit emergent behavior. Traditional V&V approaches cannot replicate the full range of possible and unexpected interactions. As a consequence, although those approaches may be able to validate that the correct system was constructed, they cannot adequately verify that the system is correct in all circumstances.

Another limitation of traditional V&V approaches is that they focus very heavily on the end product, namely, the integrated system (or subsystem). It is very well established that finding product defects late in the development cycle (or worse, after delivery) is tremendously more expensive, especially for complex systems, than correcting errors as they are made in earlier phases of development [23].

D. Failing to Recognize the Challenges

Complex aerospace systems are not simply rockets or airplanes; they are assemblies of subsystems combined with software. The human element also must be considered; as Madni notes, "the need for systems to become increasingly more adaptive to cope with changes in the operational environment has made the integration of humans with software and systems even more challenging" [24]. He goes on to say that poorly designed automation can lead to unintended changes in human performance, to the overall detriment of the performance of the system. In other words, the software may function perfectly in accordance with its requirements yet still lead to problems once a human being is introduced during system operation.

Metrics and measures are a time-honored approach for managing projects of all sizes. However, measuring software development often presents challenges, particularly for large, complex programs. For example, many U.S. government projects use earned value management (EVM) as a means of tracking program progress. EVM involves integrating project scope, schedule, and cost into a baseline plan and then tracking deviations from that baseline [25]. EVM helps project managers to answer several essential questions [26]:

1) What is the value of the work planned?
2) What is the value of the work accomplished?
3) How much did the work cost?
4) What was the total budget?
5) What do we now expect the total job to cost?

When applied correctly, EVM supports (but does not guarantee) timely, fact-based decisions and provides early warning of project problems [27,28]. Unfortunately, large aerospace projects involving software too often do not apply EVM correctly. A common error is to bury software at the lower levels of the work breakdown structure (WBS) while managing the project at higher levels. Many managers look for variations in their projects' top-line cost and schedule performance. However, detail becomes lost as one rolls up information to successively higher levels of the WBS, and a project can appear to be on track at the top level until it goes very wrong indeed. Although one expects lower-level managers to report problems as they become apparent, human nature is such that managers are frequently overly optimistic about the prospects for correction and tend to delay reporting bad news. Metrics and measures are effective only if people use them correctly to monitor progress (or, even better, to predict success) and make decisions accordingly.

Further confounding software measurement is the evolution of new development paradigms. Just as the assembly of software from already developed components introduces challenges for software architecture, so too does it challenge software development metrics. Traditional methods for measuring software progress deal poorly with the modern environment because they rely on software size (typically measured in source lines of code) as a driving factor. The size of a readily available component has no bearing on the overall development effort, yet the effect of its inclusion in the program is far from nil. Indeed, although commercial off-the-shelf (COTS) software components often are seen as a means of reducing development cost, they often bring with them other costs during system integration and testing. Esker et al. note especially the lack of benchmarks for measuring productivity and quality in systems containing such components [29]. As depicted in Fig. 5, determining the size, quality, and completeness of the software in a complex aerospace system at any given time t during the development cycle can be problematic. COTS packages and reused software may be complete at the outset of development but offer little insight as to the work that lies ahead to integrate them or how well, once combined into the final system, they will address the overall system requirements. One can track the growth of custom developed software over time, but it can be tricky to gauge its progress toward completion because software size does not necessarily correlate directly with the achievement of completeness.

Evolving development approaches also introduce new measurement concepts altogether. For instance, notions such as velocity (amount of working software delivered in a given development period), sprint burndown (daily progress toward completing a fixed workload), and release burnup (accumulation of completed work) augment traditional counts of software size, number of requirements satisfied, count of defects detected, and so on [30].

For metrics and measurement to be effective tools for managing large, complex projects involving software, one must apply those appropriate for a given project in the correct manner and then use the results to make timely decisions.

Developing large, complex systems is very hard, multidisciplinary work. Yet senior engineers and decision makers often view a system principally through the lens of their primary discipline. Thus, perhaps the most common of all root causes is the willingness of program managers and system engineers to minimize, or even simply ignore, the software aspects of the system. Software engineers often are absent from discussions about partitioning functionality, selecting computational infrastructure and communications topologies, and even early maintenance and sustainment planning. It is still fairly common for program managers and system engineers to misunderstand the detailed processes involved in software development and to question any activity that does not directly involve active coding.[‡‡] Also common is a tendency to trivialize the software effort; when an unexpected hardware or subsystem problem pops up, it is not at unusual to hear management say, "We'll fix it in software." The seeming simplicity of making a software change can be beguiling, masking the risks of creating an inconsistency in the implementation logic or introducing unintended side effects.

[‡‡]Questions about software engineers, such as "Why aren't they coding yet?" and "What do they do when they're not coding?", can still be heard in many program offices.

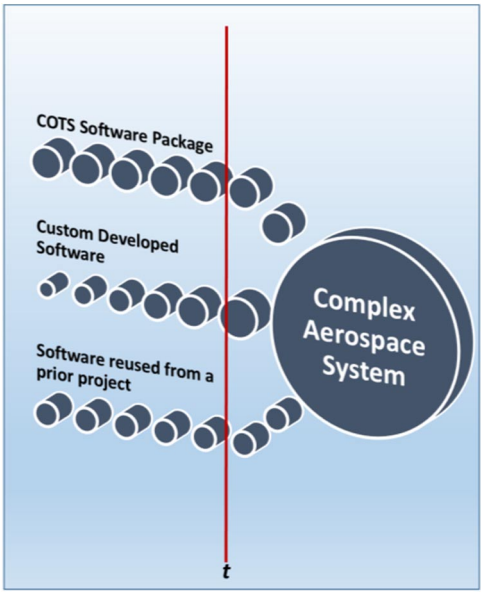


Fig. 5 Bigger? Better? Done? How can you tell?

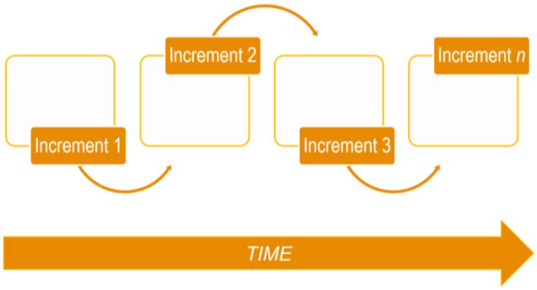


Fig. 6 Incremental development emphasizes smaller, more frequent releases of software.

IV. Overcoming the Patterns

The potentially serious impacts of software on systems, from significant delays to elevated costs to loss of equipment and life, suggest that program managers and system engineers can ill afford to ignore the software aspects of the systems they develop or procure. Yet, as demonstrated in the previous section, consideration of the challenges posed by software in a system is routinely put off. We can do better.[§§]

A. Requirements Development and Management

Identification of system quality attributes as part of the overall requirements development process can pay dividends in a number of ways. First, the quality attribute requirements serve to drive the choices of the software architects. For instance, there is often a tension between security and performance where increasing the former tends to impinge on the latter. The software architect can better balance tradeoffs knowing that both qualities are important in the system. Second, the quality attributes represent the user viewpoint during system development. Absent any other specific guidelines, engineers tend to focus on function, which is important; all users want a system that works from a functional point of view. However, whether or not they say so, users also want a system that meets their expectations in terms of timeliness of response, freedom from crashes, protection of sensitive data, and so on. Any system that fails to meet these expectations will not be well accepted by its user community. Third, techniques for divining, qualifying, and quantifying quality attributes often lead to interesting discussions among system stakeholders that surface new (or new understanding of) functional requirements.[¶¶] Barbacci et al. describe a method for discovering quality attributes and ranking them according to collective stakeholder priorities to drive software architecture development [31].

Bidirectional traceability (throughout the technical baseline) also is critical. To ensure that they are being addressed, every high-level requirement should have at least one child requirement. Likewise, every child requirement should have at least one parent requirement to ensure that the system scope has not been expanded inadvertently. Most modern requirements management tools provide a means of tracking traceability.

Program leadership must manage requirements (including quality attribute requirements) carefully, recognizing that there is a tradeoff to be made with each change to balance cost, schedule, and technical risks. Change is unavoidable but it need not lead to chaos. Requirements changes should be planned and managed to minimize perturbations to program execution. Scenarios, user stories, and other material that help elaborate the meaning of the requirements (but are not actual textual requirements in and of themselves) also should be managed to ensure they remain consistent with the requirements. Inconsistency will lead to confusion and error later in the program if allowed to seep into the technical baseline.

One approach to managing requirements effectively is through planned incremental development of software. Rather than developing software in a few large-scale releases, incremental methods focus on smaller but more frequent increments. Figure 6 illustrates the concept. Each increment is planned to implement a limited but functionally relevant subset of the overall requirements, small enough to be completed reliably on schedule. The rapid development cycle allows users to discover likes and dislikes earlier, and those opinions can then be fed back into the development

[§§]The specific techniques and methods mentioned in this section are intended to demonstrate the art of the possible. No endorsement is expressed or implied.
[¶¶]Conversations such as "That's not a requirement!" countered immediately by "Yes, it is!" are quite common.

pipeline as scheduled work in a future increment. Since the amount of new development done in each increment is relatively small, accommodating changes such as these becomes much easier than had the software been substantially completed before its first debut. In this way, incremental development allows software developers to embrace change while also controlling it. Techniques such as Scrum, extreme programming, adaptive software development, and others, which collectively are referred to as agile software development methods, are a class of incremental approaches that have received much press in software engineering circles for their ability to cope with change throughout system development. Life-cycle development models such as the Spiral Model and the Incremental Commitment Model help encourage iterative and incremental development, particularly on large programs.

B. Software Architecture Design and Evaluation

The discussion in Sec. III motivated (hopefully) the need to develop the software architecture as part of the overall system architecting process.

Having developed the software architecture, it is then important to evaluate that architecture. Only upon evaluation can program leaders have confidence that the software architecture in fact harmonizes with the system architecture and is capable of supporting all the software design to follow. Evaluation of the architecture should focus on how the architectural elements support the important system quality attributes and the engineering tradeoffs made by the architect in satisfying those attributes. Other considerations include traceability to the requirements and relative completeness. The architecture is never finished while the system is in development; it must evolve as the system does. Ideally, the evaluation of a software architecture should be conducted by independent architecture professionals in the presence of the software architect and other system stakeholders. As with other types of peer review, defects (e.g., errors and omissions), risks, questions, and action items should be recorded and tracked to closure.

Recently, advanced approaches to architecture modeling have been created to cope with the challenges of developing complex systems. Falling under the general rubric of "model-based engineering", such approaches allow prediction of a system's quality attributes because the architectural models built capture both the static and dynamic aspects of the system. Feiler and Gluch note that using model-based approaches greatly improves confidence that a system, including the software, will meet its goals [32].

C. Holistic Software Validation and Verification

It is still fairly typical to view V&V as equating to testing. As noted earlier, doing so is an expensive proposition; allowing errors to propagate through multiple work products expands the chain of rework that must be completed to address problems. A better approach is to view V&V as a process that spans the development cycle. Many organizations already engage in some form of peer review for intermediate work products, which is a good start. However, they frequently fail to consider all the opportunities for review and evaluation before the testing phase. A variety of V&V methods is essential because different techniques will detect different classes of defects at different points during development [23].

As noted in Fig. 7, more can be done throughout system development to drive defects out of the final software product at the earliest opportunity. Architecture evaluations, as discussed earlier, provide insight into the adequacy of the software architecture to meet system goals before a line of code is written. Model-based engineering approaches provide opportunities to exercise the dynamic properties of a system and its software before coding begins. Static code analysis tools can discover certain classes of problems in the software without actually executing it. Taken together, these techniques (and others) provide a holistic approach to V&V This is not to say that all of these techniques should be applied to every project. Rather, program managers should consider the overall V&V approach, in consultation with the software experts, during the program planning process to determine which techniques would most prudently be applied.

Some aspects of complex aerospace systems may be difficult, even impossible, to check via traditional V&V methods. For instance, demonstrating that a system is safe without jeopardizing a human life is a challenge. In such cases, alternate techniques are needed. The aerospace community uses the notion of a safety case to argue that a system is reliably safe. This notion can be generalized as an assurance case for similarly arguing about many properties of the software. An assurance case is nothing more than a structured argument, where evidence supports claims and ultimately justifies belief in an overarching assertion. Figure 8 depicts the basic structure of a software assurance case. A claim is an assertion about some aspect of the software or system. Evidence consists of substantiating documentation or facts. Although evidence should be objective, subjective evidence can be used if it is backed up by other objective sources. One argues up from the evidence in a logical manner to draw conclusions about the validity (or invalidity) of the primary claim [33].

Emergent behavior poses a huge challenge to V&V in complex aerospace systems; there may be no verification methods capable of detecting such behavior, whether that behavior is desired or undesired [22]. Thus, a potential means of dealing with such a V&V challenge is oriented more toward tolerance during operation than detection during development. The concept of resilience, as applied to systems and software, is exactly

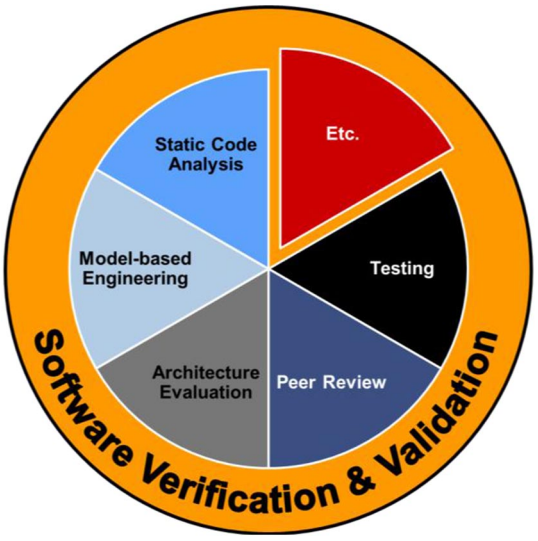


Fig. 7 A broader, holistic view of software verification and validation.

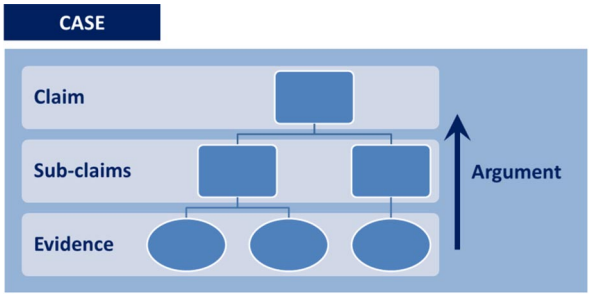


Fig. 8 The generic structure of a software assurance case.

that: designing systems and software to detect the unexpected during runtime, adapt or adjust as necessary, and continue functioning (albeit potentially in a degraded mode).

D. Project Management

Above all, sound management practices are a vital part of ensuring success for large, complex aerospace projects that rely on software. Program managers and system engineers must step out of their comfort zones and appreciate the opportunities and challenges brought to the effort by software.

Aligning life-cycle models is a good place to start. Many programs will use a more-or-less waterfall model for the overall engineering effort, whereas the software team implements an iterative approach with several partial implementations (usually called builds) introducing greater functionality over time. This mismatch causes much time and effort to be spent on figuring out how to synchronize milestones. The need to synchronize argues strongly for software management to be a participant in the overall project planning activities.

Often, projects attempt to achieve synchronization and gauge technical progress through capstone events, such as a preliminary design review or critical design review, intended to demarcate the end of one phase of development and the beginning of the next. Although well intentioned, such reviews infrequently provide sufficient alignment or technical insight and instead serve as schedule milestones; detailed technical assessment often is not possible due to voluminous information being presented to every stakeholder all at once with little opportunity for reasoned discourse. A better approach is to instantiate independent, evidence-based evaluations of progress in smaller chunks. Figure 9 depicts such a process, where one or more analysis teams composed of independent experts assess individual aspects of the development (say, an architecture evaluation, for example). The results of these assessments, including risks and proposed mitigations, can then feed into formal technical reviews as entrance criteria; the formal reviews are held in abeyance until the entrance criteria are met. Program leadership uses the exit criteria from the formal technical reviews to decide whether to proceed or to revisit certain aspects of the development deemed too risky to justify moving ahead.

One approach that embraces the need for better synchronization across program elements and evaluation of technical progress is the incremental commitment model (ICM). ICM recognizes the different strengths and challenges of hardware, software, and human factors engineering approaches and organizes their respective activities is such a way so as to maximize coordination among them while still embracing concurrent engineering principles. It is a risk-driven approach, in that it uses synchronization milestones to take a strategic pause and assess the risk of moving forward. The synchronization milestones, known as anchor points, focus on critical, evidence-based evaluation of the work to date and the plans for moving ahead, at which point program stakeholders can determine the feasibility and relative risk of continuing to the next phase of development (or of extending the current phase to address shortcomings that place unacceptable limits on the chances for success) [34]. The powerful elements of ICM are its explicit goal of coordination across engineering functions instead of disconnected system and software engineering efforts; its focus on independent, evidence-based evaluation of progress rather than reliance on schedule-driven, perfunctory reviews; and its emphasis on risk-based decision making. The reader also will note that it meshes well with recommendations in the previous sections of this paper.

Beyond aligning technical development and holding meaningful periodic reviews, project managers and systems engineers should make better use of metrics to gauge progress on a regular basis. For instance, correct use of earned value can go a long way toward providing crucial insights, especially when combined with other traditional management analysis techniques [35]. To ensure proper visibility, it is important that riskier elements of the project, such as the software effort, are not buried at the lowest levels of the WBS. Additionally, the software activities should be divided into discrete segments with defined work products so that earned value can be confidently computed; using a level-of-effort approach will hide problems until it is too late.

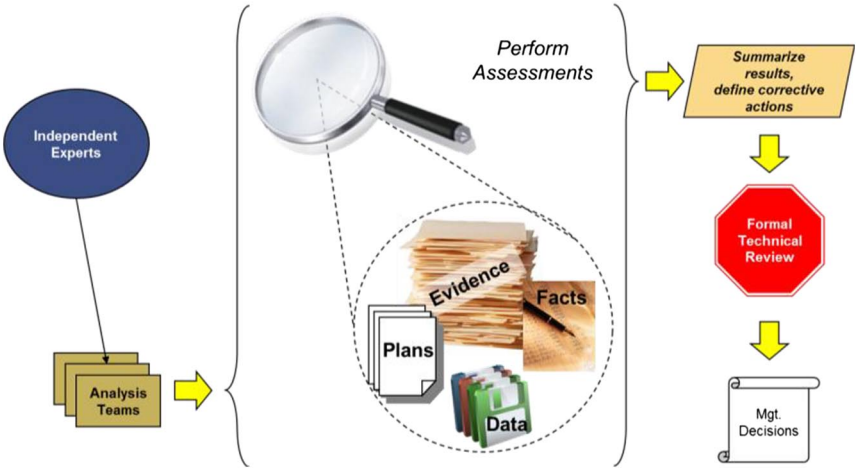


Fig. 9 Evidence-based reviews support higher confidence in technical progress.

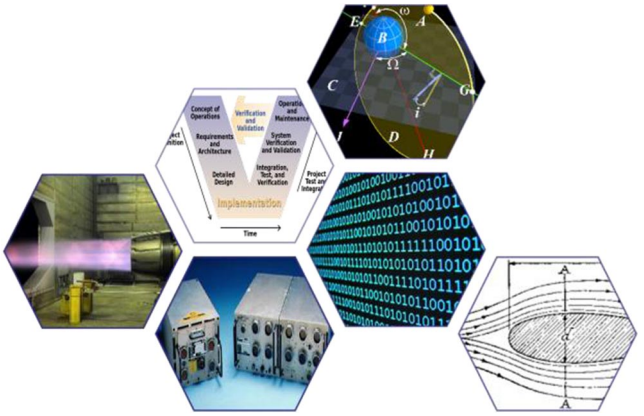


Fig. 10 Software, computers, and systems engineering must become a central part of a well-rounded aerospace curriculum.

When it comes to COTS and other readily available software components, metrics have tended to focus on either security and acquisition considerations or on the effort to develop so called "glueware", the software needed to integrate (i.e., glue) COTS components with the custom developed software [25]. Although such metrics certainly are important, they do not paint the full picture. Viewing COTS as a system integration challenge may point the way to better metrics; studies suggest that focusing on report, interface, conversion, and extension objects (the so-called RICE objects) may lead to development of appropriate metrics for development efforts that include COTS [29,36].

V. Longer Term: A View Toward Prevention

Recognizing that success of software had become a fundamental underpinning of its own success, NASA embarked on a multiyear study to identify and codify software engineering best practices for its projects. The study, which ran from 2010 through 2013, culminated in a set of 14 suggested recommendations to combat some of the agency's most vexing software engineering issues and improve the state of its software engineering practices overall [37].

In the final analysis, however, none of the preceding measures, or any others, is likely to be applied with any consistency unless program managers and system engineers either are aware of them or are willing to be counseled by their software engineering colleagues. Therein lies the rub, for the aerospace and defense industry suffers from an educational pipeline that has not kept pace with the rapid technological changes happening across the broader commercial world.

Aerospace education, in large measure, still tends to focus on its traditional underpinnings: aerodynamics, structures, propulsion, and control. Long notes that, although aerospace engineers have long prided themselves on being integrators, their ability to do so effectively for the large, complex systems of today is compromised by shortfalls in their education, particularly in the areas of computers, networking, sensors, and software [4]. He goes on to say that most aerospace engineering programs mandate only one freshman-level programming class in a four-year course of study and, although that single class may yield some small appreciation for the difficulties of writing code for a relatively trivial assignment, the other subtleties of software engineering throughout the development life cycle likely are absent. Jolly echoes the concern about integration, highlighting the folly of simply procuring system components without understanding "their design, their failure modes, their interaction with the physical spacecraft and its environment, and how *software* [emphasis added] knits the whole story together" [38]. Simply put, today's aerospace education programs, by and large, leave students with a critical gap in their understanding of the systems they will go on to design and build. When one considers that today's aerospace engineers become tomorrow's system engineers and program managers, the seriousness of the knowledge gap becomes apparent.

For their part, software engineering programs are not much better, typically turning out students who are generalists able to deal with a wide range of software development applications but often lacking some of the fundamental engineering knowledge that is the bedrock of other engineering disciplines.[***] Further, software engineers often struggle to communicate effectively with their other engineering counterparts, preferring the insular comforts of their own jargon and culture that make it hard for anyone else to understand them, let alone take them seriously. Indeed, Jolly, a veteran space system engineer, even has called for the transformation of the systems, software, and avionics engineering disciplines in the aerospace community [38].

Building on these notions, Fig. 10 argues that a complete, well-rounded aerospace curriculum ought to include study in software, computers, and systems engineering in addition to the classical aerospace engineering subjects. However, the prospects for updated aerospace curricula are uncertain. Many traditional universities employ a slow and deliberative process for modifying curricula, putting them at odds with the pace of change in the Information Age [39]. Additionally, many aerospace engineering programs are reticent to add credit hours to already demanding courses of study, while simultaneously finding the thought of substituting needed software and computing education for customary but perhaps less useful coursework untenable.

The main accreditation body for aerospace education (and, indeed, most engineering curricula), ABET, contributes to the inertia, stating, "Aeronautical engineering programs must prepare graduates to have a knowledge of aerodynamics, aerospace materials, structures, propulsion, flight mechanics, and stability and control. Astronautical engineering programs must prepare graduates to have a knowledge of orbital mechanics, space environment, attitude determination and control, telecommunications, space structures, and rocket propulsion" [40]. Without recognition of software, computing, or even systems engineering as part of the ABET criteria for aerospace education, there will be little motivation on the part of universities to adapt their current programs.[†††] Note that ABET does offer guidance on systems engineering programs specifically, saying, "There are no program-specific criteria beyond the General Criteria" [40].

The National Academy of Engineering has remarked upon the knowledge gap more generally across disciplines: current undergraduate engineering programs in the U.S. simply are not preparing young engineers for the challenges they will face in the coming years. Among its many recommendations, the Academy suggests the importance of interdisciplinary studies and lifelong learning to the ultimate success of engineering

***Indeed, there is much debate about whether software engineering is really an engineering discipline at all.

†††Interestingly, "telecommunications" at least receives a nod as being important in astronautical engineering programs but, dichotomously, not in aeronautical engineering programs.

students as practicing professionals [41]. In that light, it seems entirely reasonable to propose that current system engineers and program managers expand their understanding of software, that software engineers working on aerospace systems expand their understanding of such systems, and that programs overall embrace the multidisciplinary challenges of developing large, complex aerospace systems. Only then can we be justifiably confident in successful outcomes.

VI. Conclusions

Software has become, and will continue to be, a significant component of all large, complex aerospace systems. Despite numerous examples of problems, the aerospace industry fails to adapt its processes, techniques, and, most importantly, its thinking to include software considerations throughout the system development cycle. Consequently, the industry leaves itself open to preventable catastrophic failures induced by software. To prevent software from becoming a giant slayer, different approaches are needed. Changing some of the risky, repeating patterns of program execution would be a welcome step toward improving the chances for successful outcomes. In particular, more integrated approaches to program management and system engineering overall, ones that includes appropriate consideration of the software aspects of a system and corresponding development techniques, are called for.

Over the long term, ensuring that such changes are embraced may ultimately depend on a shift in engineering education, across disciplines, to reflect the interdisciplinary complexity of the systems being developed now as well as those not yet imagined. However, for a variety of reasons, such changes likely will be slow to occur.

Although modernization of the educational system may be a long-term solution, current practitioners certainly can, and should, begin to broaden their own thinking. The approaches mentioned herein represent only a few of the possibilities for better, holistic management and control of large, complex aerospace projects. Adopting even some of them will help ensure that software continues enabling the complex aerospace systems of tomorrow instead of killing them.

References

[1] Dvorak, D. L., (ed.), *NASA Study on Flight Software Complexity*, NASA, 2009.

[2] Hagen, C., Sorenson, J., Hurt, S., Heckler, A., and Wall, D., "Software: The Brains Behind US Defense Systems," A. T. Kearney, Arlington, VA, 2012.

[3] "Report of the Defense Science Board Task Force on Defense Software," Office of the Under Secretary of Defense for Acquisition and Technology, Washington, D.C., Nov. 2000.

[4] Long, L., "The Critical Need for Software Engineering Education," *CrossTalk*, Vol. 21, No. 1, Jan. 2008, pp. 6–10.

[5] *Defense Acquisition Guidebook*, U.S. Dept. of Defense, Washington, D.C., 2013, Chap. 4.

[6] "F-35 Joint Strike Fighter: Problems Completing Software Testing May Hinder Delivery of Expected Warfighting Capabilities," Government Accountability Office, Rept. GAO-14-322, Washington, D.C., March 2014.

[7] Schumann, J., Mbaya, T., Mengshoel, O., Pipatsrisawat, K., Srivastava, A., Choi, A., and Darwiche, A., "Software Health Management with Bayesian Networks," *Innovations in Systems and Software Engineering*, Vol. 9, No. 4, Dec. 2013, pp. 271–292. doi:10.1007/s11334-013-0214-y

[8] Srivastava, A., and Schumann, J., "The Case for Software Health Management," *Proceedings of the 2011 IEEE 4th International Conference on Space Mission Challenges for Information Technology*, IEEE Publ., Piscataway, NJ, 2011, p. 6. doi:10.1109/SMC-IT.2011.14

[9] "Crash of Asiana Flight 214 Accident Report Summary," National Transportation Safety Board, June 2014.

[10] "Human Performance Issues, Asiana Flight 214, B777," National Transportation Safety Board, Senior Human Performance Investigator, June 2014.

[11] Sherry, L., and Mauro, R., "Controlled Flight into Stall (CFIS): Functional Complexity Failures and Automation Surprises," *Proceedings of the Integrated Communications, Navigation and Surveillance Conference (ICNS)*, IEEE Publ., Piscataway, NJ, 2014, pp. 1–0. doi:10.1109/ICNSurv.2014.6819980

[12] "In-Flight Upset, 154 km West of Learmonth, Western Australia, 7 October 2008, VH-QPA, Airbus A330-303," Australian Transport Safety Bureau, Rept. AO-2008-070, Canberra, Australia, Dec. 2011.

[13] "Airworthiness Directives; The Boeing Company Airplanes," Federal Aviation Administration, Rept. AD 2014-06-04, Washington, D.C., April 2014.

[14] "Special Conditions: Boeing Model 787-8 Airplane," Federal Aviation Administration, Special Conditions No. 25-356-SC, Washington, D.C., 2008.

[15] "Special Conditions: Boeing Model 777- 200, -300, and -300ER Series Airplanes," Federal Aviation Administration, Rept. 25–503–SC, Washington, D.C., Nov. 2013.

[16] Guzzetti, J., and Langan-Feirson, M. K., "Weaknesses in Program and Contract Management Contribute to ERAM Delays and Put Other NextGen Initiatives at Risk," Federal Aviation Administration Paper AV-2012-179, Washington, D.C., Sept. 2012.

[17] Dowson, M., "The ARIANE 5 Software Failure," *Software Engineering Notes*, Vol. 22, No. 2, March 1997, p. 84. doi:10.1145/251880

[18] Lions, J. L., "ARIANE 5 Flight 501 Failure Report by the Inquiry Board," Paris, France, July 1996.

[19] Croll, P. R., "Quality Attributes: Architecting Systems to Meet Customer Expectations," *Proceedings of the IEEE International Systems Conference Proceedings*, IEEE Publ., Piscataway, NJ, 2008, pp. 56–63. doi:10.1109/SYSTEMS.2008.4518988

[20] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, 3rd ed., Addison-Wesley, Boston, MA, 2012, Chap. 1.

[21] Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, Boston, MA, 2002, Chap. 1.

[22] Morris, D. M., and Adams, K. M., "The Whole is More than the Sum of Its Parts: Understanding and Managing Emergent Behavior in Complex Systems," *CrossTalk*, Vol. 26, No. 1, Jan.–Feb. 2013, pp. 9–14.

[23] Boehm, B., and Basili, V. R., "Software Detect Reduction Top 10 List," *Computer*, Vol. 34, No. 1, Jan. 2001, pp. 135–137. doi:10.1109/2.962984

[24] Madni, A. M., "Integrating Humans with and Within Complex Systems," *CrossTalk*, Vol. 24, No. 3, May–June 2011, pp. 4–8.

[25] Staley, M. J., Oberndorf, P., and Sledge, C. A., "Using EVMS with COTS-Based Systems," Software Engineering Inst., Rept. CMU/SEI-2002-TR-022, Pittsburgh, PA, June 2002.

[26] Brownsword, L., and Smith, J., "Using Earned Value Management (EVM) in Spiral Development," Software Engineering Inst., Rept. CMU/SEI-2005-TN-016, Pittsburgh, PA, June 2005.

[27] Garrett, G. A., "Earned Value Management," *Contract Management*, Vol. 46, No. 12, Dec. 2006, pp. 49–51.

[28] Fleming, Q. W., and Koppelman, J. M., "Earned Value Project Management," *CrossTalk*, Vol. 11, No. 7, July 1998, pp. 19–23.

[29] Esker, L., Diep, M., and Herman, F., "Seeking Meaningful Measures for COTS-Intensive System, Development," *AIAA Infotech@Aerospace Conference*, AIAA Paper 2015-1646, 2015. doi:10.2514/6.2015-1646

[30] Hayes, W., Miller, S., Lapham, M. A., Wrubel, E., and Chick, T., "Agile Metrics: Progress Monitoring of Agile Contractors," Software Engineering Inst., Rept. CMU/SEI-2013-TN-029, Pittsburgh, PA, Jan. 2014.

[31] Barbacci, M. R., Ellison, R., Lattanze, A. J., Stafford, J. A., Weinstock, C. B., and Wood, W. G., *Quality Attribute Workshops (QAWs)*, 3rd ed., Software Engineering Inst., CMU/SEI-2003-TR-016, Pittsburgh, PA, Aug. 2003.

[32] Feiler, P. H., and Gluch, D. P., *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, Addison-Wesley, Upper Saddle River, NJ, 2013, p. 1.
[33] Blanchette, S., Jr., "Assurance Cases for Design Analysis of Complex System of Systems Software," *AIAA Infotech@ Aerospace Conference*, AIAA Paper 2009-1921, April 2009.
doi:10.2514/6.2009-1921
[34] Boehm, B., and Lane, J. A., "Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering," *CrossTalk*, Vol. 20, No. 10, Oct. 2007, pp. 4–9.
[35] "Earned Value Management Systems, Government Electronics and Information Technology Association," Electronic Industries Alliance, Rept. ANSI/EIA-748-B-2007, Arlington, VA, 2007, p. 18.
[36] Rosa, W., Packard, T., Krupanand, A., Bilbro, J. W., and Hodal, M. M., "COTS Integration and Estimation for ERP," *Journal of Systems and Software*, Vol. 86, No. 2, Feb. 2013, pp. 538–550.
doi:10.1016/j.jss.2012.09.030
[37] Rarick, H. L., Godfrey, S. H., Kelly, J. C., Crumbley, R. T., and Wilf, J. M., "NASA Software Engineering Benchmarking Study," NASA SP-2013-604, May 2013.
[38] Jolly, S., "Is Software Broken?" *ASK Magazine*, No. 34, Sept. 2009, pp. 22–25.
[39] Long, L. N., "On the Need for Significant Reform in University Education, Especially in Aerospace Engineering," *Proceedings of the IEEE Aerospace Conference*, IEEE Publ., Piscataway, NJ, 2015, pp. 1–7.
doi:10.1109/AERO.2015.7119066
[40] "Criteria for Accrediting Engineering Programs," Engineering Accreditation Commission, Baltimore, MD, 2013, pp. 6, 21.
[41] *Educating the Engineer of 2020: Adapting Engineering Education to the New Century*, National Academies Press, Washington, D.C., 2005, p. 55.

M. D. Davies
Associate Editor