INSIGHT

This Issue's Feature:

Systems and Software

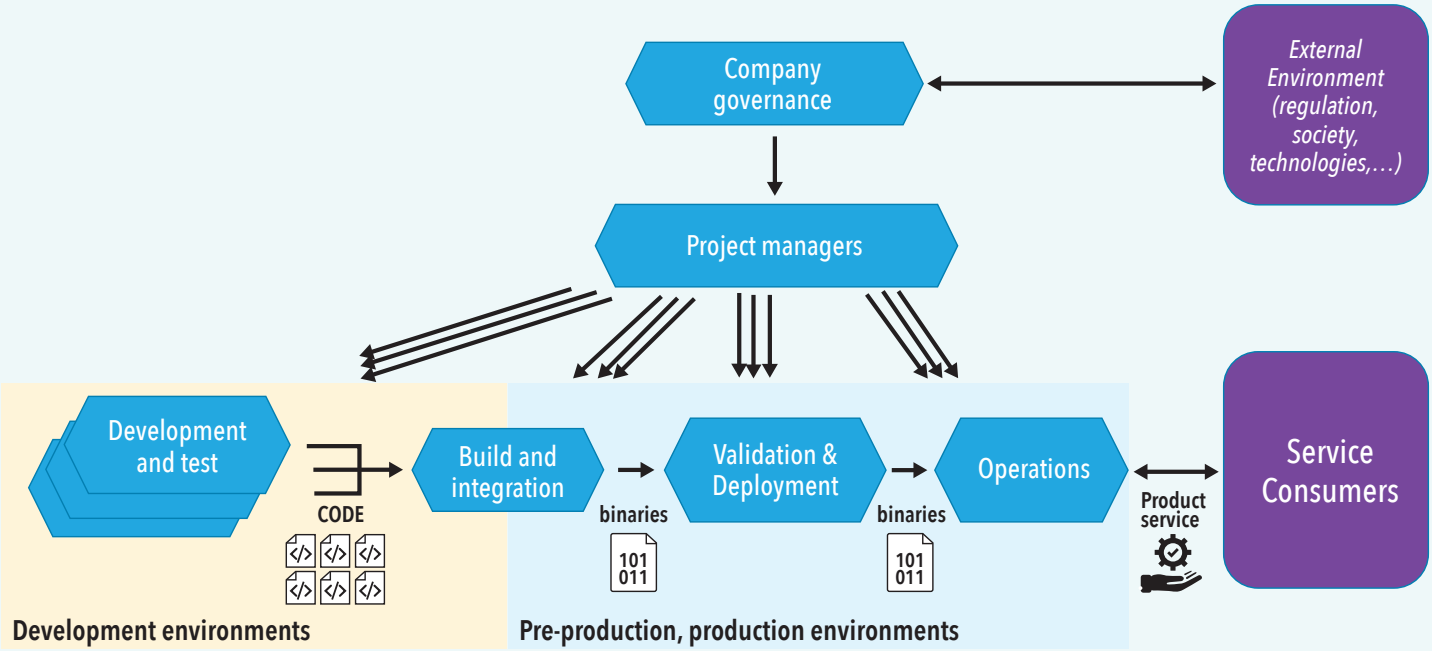


Illustration credit: from the article
Systems Engineering Roles in Software Organizations Delivering Service Products
by Mickael Bouyaud and Brian E. White (see page 21)

JULY 2021
VOLUME 24 / ISSUE 2

A PUBLICATION OF THE INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING

INCOSE®

INSIGHT

A PUBLICATION OF THE INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING

JULY 2021 VOLUME 24 / ISSUE 2

Inside this issue

About This Publication

INFORMATION ABOUT INCOSE

INCOSE's membership extends to over 18,000 individual members and more than 100 corporations, government entities, and academic institutions. Its mission is to share, promote, and advance the best of systems engineering from across the globe for the benefit of humanity and the planet. INCOSE charters chapters worldwide, includes a corporate advisory board, and is led by elected officers and directors.

For more information, click here:
The International Council on Systems Engineering
(www.incose.org)
INSIGHT is the magazine of the International Council on Systems Engineering. It is published four times per year and

OVERVIEW

features informative articles dedicated to advancing the state of practice in systems engineering and to close the gap with the state of the art. ***INSIGHT*** delivers practical information on current hot topics, implementations, and best practices, written in applications-driven style. There is an emphasis on practical applications, tutorials, guides, and case studies that result in successful outcomes. Explicitly identified opinion pieces, book reviews, and technology roadmapping complement articles to stimulate advancing the state of practice.
INSIGHT is dedicated to advancing the INCOSE objectives of impactful products and accelerating the transformation of systems engineering to a model-based discipline.
Topics to be covered include resilient systems, model-based systems engineering, commercial-driven transformational systems engineering, natural systems, agile security, systems of systems, and cyber-physical systems across disciplines and domains of interest to the constituent groups in the systems engineering community: industry, government, and academia. Advances in practice often come from lateral connections of information dissemination across disciplines and domains. ***INSIGHT*** will track advances in the state of the art with follow-up, practically written articles to more rapidly disseminate knowledge to stimulate practice throughout the community.

ARTICLE SUBMISSION

insight@incose.net

Publication Schedule. ***INSIGHT*** is published four times per year. Issue and article submission deadlines are as follows:

- December 2021 issue – 1 October 2021
- March 2022 issue – 2 January 2022
- June 2022 issue – 1 April 20022
- September 2022 issue – 1 July 2022

For further information on submissions and issue themes, visit the INCOSE website: www.incose.org

ISSN 2156-485X; (print) ISSN 2156-4868 (online)

ADVERTISE

Readership
INSIGHT reaches over 18,000 individual members and uncounted employees and students of more than 100 CAB organizations worldwide. Readership includes engineers, manufacturers/purchasers, scientists, research & development processionals, presidents and CEOs, students and other professionals in systems engineering.

Issuance	Circulation
2021, Vol 24, 4 Issues	100% Paid

Contact us for Advertising and Corporate Sales Services
We have a complete range of advertising and publishing solutions professionally managed within our global team. From traditional print-based solutions to cutting-edge online technology the Wiley-Blackwell corporate sales service is your connection to minds that matter. For an overview of all our services please browse our site which is located under the Resources section. Contact our corporate sales team today to discuss the range of services available:

- Print advertising for non-US journals
- Email Table of Contents Sponsorship
- Reprints
- Supplement and sponsorship opportunities
- Books
- Custom Projects
- Online advertising

Click on the option below to email your enquiry to your nearest office:

- Asia & Australia corporatesalesaustralia@wiley.com
- Europe, Middle East & Africa (EMEA) corporatesaleseurope@wiley.com
- Japan corporatesalesjapan@wiley.com
- Korea corporatesaleskorea@wiley.com

USA (also Canada, and South/Central America):

- Healthcare Advertising corporatesalesusa@wiley.com
- Science Advertising Ads_sciences@wiley.com
- Reprints Commercialreprints@wiley.com
- Supplements, Sponsorship, Books and Custom Projects busdev@wiley.com

Or please contact:
Susan Blessing, Senior Account Manager Sciences
Sciences, Corporate Sales
MOBILE: 24/7 201-723-3129
E-MAIL: sblessin@wiley.com

CONTACT

Questions or comments concerning:

Submissions, Editorial Policy, or Publication Management
Please contact: William Miller, Editor-in-Chief
insight@incose.net

Advertising—*please contact:*
Susan Blessing, Senior Account Manager Sciences
Sciences, Corporate Sales
MOBILE: 24/7 201-723-3129
E-MAIL: sblessin@wiley.com

Member Services – *please contact:* info@incose.net

ADVERTISER INDEX

July volume 24-2

CORPORATE ADVISORY BOARD — MEMBER COMPANIES

321 Gang, Inc.
Aerospace Corporation, The
Airbus
AM General LLC
Analog Devices, Inc.
Aras Corp
Australian Naitonal University
Aviation Industry Corporation of China, LTD
BAE Systems
Ball Aerospace
Bechtel
Beckton Dickinson
Blue Origin
Boeing Company, The
Bombardier Transportation
Booz Allen Hamilton Inc.
C.S. Draper Laboratory, Inc.
California State University Dominguez Hills
Carnegie Mellon University Software Engineering Institute
Change Vision, Inc
Colorado State University
Cornell University
Cranfield University
Cubic Corporation
Cummins, Inc.
CYBERNET MBSE
Defense Acquisition University
Deloitte Consulting
DENSO Create, Inc.
Drexel University
Eindhoven University of Technology
Embraer S.A.
ENAC
Federal Aviation Administration (U.S.)
Ford Motor Company
Fundacao Ezute
General Dynamics Mission Systems
General Electric Aviation
General Motors
George Mason University
Georgia Institute of Technology
IBM
Idaho National Laboratory
ISAE SUPAERO
ISDEFE
iTiD Consulting, Ltd
Jacobs Engineering
Jama Software
Jet Propulsion Laboratory
John Deere & Company
Johns Hopkins University
KBR, Inc.
KEIO University
L3 Harris
Leidos
Lockheed Martin Corporation
Los Alamos National Laboratory
ManTech International Corporation
Maplesoft
Massachusetts Institute of Technology
MBDA (UK) Ltd.
Missouri University of Science & Technology
MITRE Corporation, The
Mitsubishi Heavy Industries
National Aeronautics and Space Administration
National Security Agency – Enterprise
Naval Postgraduate School
Nissan Motor Co, Ltd
No Magic/Dassault Systems
Northrop Grumman Corporation
Pacific Northwest National Laboratories
Penn State University
Peraton (formerly Perspecta formerly Vencore)
PETRONAS NASIONAL BERHAD
Prime Solutions Group, Inc.
Project Performance International
QRA Corporation
Raytheon Corporation
Roche Diagnostics
Rolls-Royce
Saab AB
SAIC
Sandia National Laboratories
Siemens
Sierra Nevada Corporation
Singapore Institute of Technology
Skoltech
SPEC Innovations
Stellar Solutions
Stevens Institute of Technology
Strategic Technical Services
Swedish Defence Materiel Administration
Systems Engineering Directorate
Systems Planning and Analysis
Thales
Torch Technologies
Trane Technologies
Tsinghua University
TUS Solution LLC
UK MoD
University of Alabama in Huntsville
University of Arkansas
University of California San Diego
University of Connecticut
University of Maryland
University of Maryland, Baltimore County
University of Michigan, Ann Arbor
University of New South Wales, The, Canberra
University of Southern California
University of Texas at El Paso, The
University of Washington, Industrial & SE Dept
US Department of Defense, Deputy Assistant Secretary of Defense for Systems Engineering,
Veoneer, Inc
Vitech Corporation
Volvo Construction Equipment
Woodward Inc
Worcester Polytechnic Institute – WPI
Zuken, Inc.

JULY 2021
VOLUME 24/ ISSUE 2

FROM THE EDITOR-IN-CHIEF

William Miller, insight@incose.org

It is our pleasure to announce the July 2021 *INSIGHT* issue published cooperatively with John Wiley & Sons as the systems engineering practitioners' magazine. The *INSIGHT* mission is providing informative articles on advancing the systems engineering practice and to close the gap between practice and the state of the art as advanced by *Systems Engineering*, the Journal of INCOSE also published by Wiley.

The issue theme is systems and software that is critical to the future of systems engineering initiative throughout the systems community. We thank theme editors Sarah Sheard and John Wood, the Systems and Software Interface Working Group (SaSIWG), and the authors for their contributions. Sarah's and John's lead article provides the context for this *INSIGHT* issue in terms of the interface between systems and software with the imperative that systems engineers and software engineers know enough about each other's fields to ensure the interface works smoothly. Their article goes on to serve the reader with a brief synopsis of each article relevant to the theme.

Your editor appreciates from professional experiences the challenges of systems and software interfaces across the complex socio-technical divide that is subject to different meanings using the same words, and divergent "facts," beliefs, and biases. As a Bell Labs systems engineer, I lived the transition of telephone systems based on electro-mechanical technologies dependent upon manual methods and procedures, to computer/software aided methods and procedures providing operations support to the telephone system, to centralized stored program control at the system level, and thence on to distributed embedded computing. I vividly remember the "radical" at the time justification for placing a 32-bit microcomputer at the circuit card level in a distributed computing telephone switch to give flexibility over the life cycle of that switch. I later served as a chief systems engineer across multiple programs in a different domain that experienced a decade long transition from an unwritten requirement that "there shall be no software in the systems" to "getting our feet wet" with software peripheral to the systems, and thence on to mission critical software embedded throughout the systems. The same story applies to the innovation of computing and software in aerospace systems as illuminated by the Apollo program and commercial air transport aircraft. In parallel I witnessed the growth of computer science and software engineering as disciplines on a par with "traditional" engineering disciplines. My observation is that the time constant for these transitions is driven by the progression of engineers' knowledge and competencies to be half a generation, that is, about 10 years. The Systems Engineering Research Center is researching how to lower the time constant for systems engineers (https://sercuarc.org/experience-accelerator/).

The patterns in systems engineering processes, methods, and tools we use have a legacy in the era where software was at best centralized, or marginally, part of the periphery of enabling systems. For example, the work breakdown structure (WBS) for aircraft and drones is documented in Mil-Std-881E dated 6 October 2020. From the standard:

> "The Program WBS and Contract WBS aid in documenting the work effort necessary to produce and maintain architectural products in a system life cycle. The DoD Architecture Framework (DoDAF) (current version) defines a common approach for DoD architecture description development, presentation, and integration for warfighting operations and business operations and processes."

Philosophically, how do these legacy processes, methods, and tools contribute to the "impedance (mis)match" inertia in the interface with software engineering? Holistically, why not envision aircraft or drones as cyber-physical systems, or more colloquially, as "flying computers" critically dependent on software to be fit for purpose and do no harm? Look at our mobile devices whose multi-purpose functionality is critically dependent on software to achieve the small form fit and smart electrical power management that integrates what were separate devices before the mid-2000s. How might that perspective influence the architecture of these systems to simplify the systems and software interface? Perhaps we should look to the systems engineering and software engineering practiced by the big five tech giants collectively known as GAFAM or FAAMG: Alphabet (Google's parent), Amazon, Apple, Facebook, and Microsoft; as well as SpaceX. Google's site reliability engineering, described both online and in print, covers both systems and software engineering.

To quote INCOSE past president John Thomas: "It's a great time to be a systems engineer!"

We hope you find *INSIGHT*, the practitioners' magazine for systems engineers, informative and relevant. Feedback from readers is critical to *INSIGHT*'s quality. We encourage letters to the editor at insight@incose.net. Please include "letter to the editor" in the subject line. *INSIGHT* also continues to solicit special features, standalone articles, book reviews, and op-eds. For information about *INSIGHT*, including upcoming issues, see https://www.incose.org/products-and-publications/periodicals#INSIGHT. For information about organizations sponsoring *INSIGHT*, please contact the INCOSE marketing and communications director at marcom@incose.net. ■

JULY 2021
VOLUME 24/ ISSUE 2

Introduction to Themed Edition

Sarah Sheard, sarah.sheard@gmail.com;and **John Wood,** woodjn@gwu.edu

In the 21st century, software has become central to business as electrical infrastructure and plumbing, but software is unique. Software

- Controls critical machine functions ranging from pacemakers to aircraft;
- Allows people and businesses to communicate across the globe;
- Provides capabilities distinguishing one business from another which can make or break businesses financially;
- Provides an unprecedented entrance for criminals to many daily life aspects.

For these reasons, systems engineers must become comfortable dealing with software in systems.

The Systems and Software Interface Working Group (SaSIWG), which Sarah Sheard chaired through the 2021 INCOSE International Workshop (IW), now chaired by Nick Guertin, works on numerous potential products to help the INCOSE systems engineers do just that.

Two SaSIWG members surveyed senior systems engineers and senior software engineers to identify pain points between the two disciplines for almost two years. These members documented the survey's results in the Working Group's INCOSE International Symposium (IS) 2020 paper, which Sarah Sheard helped write. We include it here as — Article 1, "Systems and Software Interface Survey" by Sally Muscarella, Macaulay Osaisai, and Sarah Sheard. The bottom line is systems engineering skills are still essential and must evolve to include model-based systems engineering. Both systems engineers and software engineers must learn enough about each other's fields to ensure the interface works smoothly. We include the IS paper in its entirety in this *INSIGHT* issue.

Article 2, ""Book Club Guides A Working Group to Create INCOSE System-Software Interface Products," by Sarah Sheard, Mickael Bouyaud, Macaulay Osaisai, Jeannine Siviy, and Ken Nidiffer, describes the "book club" in which we studied a popular fable-type book, *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data* by Gene Kim (originally suggested by Shirley Tseng). The book club resulted in several working group products to help us understand issues and topics arising during our book club discussions. Some products were the impetus for articles included in this issue.

Article 3, "System Engineering Roles in Software Organizations Delivering Service Products," by Mickael Bouyaud and Brian White, started as a software service delivery explanation to the book club by the SaSIWG member most knowledgeable on software service delivery. This article, which expands upon some charts from our IS 2021 paper (Sheard et al. 2021), shows how software development proceeds within a company using examples from *The Unicorn Project* (Kim 2019).

We were also lucky to have within the book club some significant complex systems talent. Article 4, "A Complex Adaptive Systems Engineering Methodology," by Brian White and Mickael Bouyaud, resulted from book club discussions about better company organization methods for addressing the rising complexity of current and future systems. The authors show how to move from today's overly simplistic management by Program Evaluation and Review Technique (PERT)-type charts to a methodology, based on evolving complexity science, more suited to manage creating complex systems of systems.

We also felt the organization in Gene Kim's book addressed, but did not talk specifically about, the DevSecOps (Secure Development and Operation) issues working together with systems engineering. Article 5, "Systems Engineering and DevSecOps," by Richard Turner, addresses these issues outright from a software engineering viewpoint. This article will be useful to systems engineers by showing how "the other side" thinks and, thus, forms the foundation upon which we need to begin our discussions. Dr. Turner lays out nine DevSecOps principles and shows how each relates to systems engineering as currently practiced in software-intensive organizations. The article also provides a helpful comparison between DevSecOps, Lean, Agile, and systems engineering principles.

Article 6, "System Test Approach for Complex Software Systems," by Chandru Mirchandani, addresses incorporating the actual operational environment into software system tests to measure software quality. While some software engineers claim software reliability is one (100%), others say removing all software defects is impossible. Others argue we can never fully test software. Even if these general statements were true, they do not provide the fidelity needed by systems engineers. Systems engineers must understand and talk about quantity and density of failures and faults as well as times between failures. Otherwise, we cannot know if the software's contribution to the overall product attributes will meet system-level requirements. This article offers systems engineers a method for calculating

requirements-based and functionality-based failure rate profiles for software-intensive systems. It also provides a starting point for discussing whether the findings are acceptable at a system level.

And finally, Article 7, "Systems Thinking and Business Resilience: Questions that Should Keep Us Up at Night," by Jeannine Siviy and Gene Kim, examines "Where do we go from here?" We need all the immense talent INCOSE has and represents to tackle the enormous system questions posed in this article. For example, into what categories can we sort different contexts? What roles should systems engineers and other systems thinkers play in the various contexts? Then, once we find out what we want to have happen, how do we make it happen?

We hope you enjoy these articles, and we hope they expand your current understanding of the interface between software engineering and systems engineering. More importantly, we hope these articles encourage systems engineers to practice their profession confidently within software-intensive organizations. We may be biased, but we believe people with a systems perspective are invaluable in helping complex organizations achieve their goals. ■

REFERENCES

- Kim, G. 2019. *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. Portland, US-OR: IT Revolution Press.
- Sheard, S., M. Bouyaud, M. Osaisai, J. Siviy, and K. Nidiffer. "A Guide for Systems Engineers to Finding Your Role in 21st Century Software-Dominant Organizations." Paper presented at the 31st Annual International Symposium of INCOSE, Virtual Event, 17–22 July.

ABOUT THE AUTHORS

Dr. Sarah Sheard is an INCOSE Fellow, CSEP, and Founder's Award winner. An INCOSE member since 1992, she chaired INCOSE's SaSIWG from 2017-2021. Her many systems engineering publications include three INCOSE "Best Papers." When she retired in 2019, Dr. Sheard was a systems and software engineering researcher and consultant at CMU's Software Engineering Institute. Prior to that time, she worked at the Systems and Software Consortium, Loral/IBM Federal Systems, and Hughes Aircraft Company. In 2012, she earned her systems engineering Ph.D. focusing on system development complexity from the Stevens Institute of Technology. Now, she postponed international travel to learn folk dancing by Zoom with her husband.

Dr. John Wood currently serves as Naval Information Warfare Center Pacific department lead systems engineer. He leads world-class engineers to create future warfighting capabilities for the US Navy and other Department of Defense customers. Dr. Wood has consistently applied his systems engineering and project management expertise to solve mission and safety critical challenges within healthcare, aviation, and defense throughout his career. Dr. Wood holds an electrical engineering B.S. from the US Naval Academy and a systems engineering Ph.D. from the George Washington University.

Systems and Software Interface Survey

Sally Muscarella, scmuscarella@gmail.com; **Macaulay Osaisai,** Macaulay.Osaisai@L3Harris.com; and **Sarah Sheard,** sarah.sheard@gmail.com

ABSTRACT
INCOSE formed the Systems and Software Interface Working Group (SaSIWG) in 2017, responding to the Corporate Advisory Board interest in software and problems identified in the systems and software interface (physical, logical, data, and human). This third SaSIWG paper presents a systems engineers, software engineers, and project managers survey discussing best practices and the priority challenges related to the interface between systems and software. We group and summarize the best practices mentioned by the 31 interviewees then address priority challenges and problems. Systems engineering done well includes an ever-increasing amount of Model-Based Systems Engineering. It also includes developing and holding to a vision, managing data, ensuring inter-disciplinary work, planning systematic verification, and ensuring modularity. Systems engineering must evolve to meet new challenges and, most importantly, systems engineer expertise must include software engineering.

INTRODUCTION

Systems engineering and software engineering grew from the physical systems world and the computer science world (Sheard, Pafford, and Phillips 2019). Since complex systems today are or may become cyber-physical systems, their development needs cyber and physical skills. Fortunately, since the 1990s, engineers of both disciplines are becoming familiar with the processes and methodologies the other discipline uses. For example, the Institute of Electrical and Electronics Engineers (IEEE) began work in the 1990s to "harmonize" systems and software standards, primarily ISO/IEC 15288 and 12207, respectively (ISO/IEC/IEEE, 2015 and 2017).

However, other sources describing failed projects frequently mention breakdowns between systems and software engineering teams or system interfaces and software integration. When asked, software engineers blame systems engineering inadequacies, and the systems engineers blame software engineering inadequacies.

In 2016, INCOSE asked Corporate Advisory Board (CAB) members to prioritize seven to 10 topics INCOSE should improve. Somewhat surprisingly, "software" was not a top issue for any company. However, nearly all companies listed it somewhere, so it became a CAB "top 7 concern." A January 2017 CAB breakout session asked CAB representatives to recommend what issues INCOSE should address in this area (Bramer 2017).

At the 2017 INCOSE International Symposium (IS), INCOSE Past President Heinz Stoewer emphasized systems-software interface's important work. He warned if INCOSE "does not address the digital thread" quickly, to understand software and start leading software-intensive systems, "INCOSE risks becoming irrelevant."

To initiate action, INCOSE organized the Systems and Software Interface Working Group (SaSIWG). The group first met during IS 2017 and based their charter (SaSIWG 2017) on the CAB results, augmented by member knowledge.

The SaSIWG presented a paper in 2018 describing the results from the initial brainstorming meeting in 2017 (Sheard et al. 2018). This included describing interface problem areas. Then in 2019, a SaSIWG paper (Sheard, Pafford, and Phillips 2019) focused on coordinating the systems engineering and software engineering discipline tasks and roles enacting a high-performance systems and software enterprise.

CURRENT SITUATION: SOFTWARE-INTENSIVE SYSTEMS CHALLENGE THE SYSTEMS AND SOFTWARE ENGINEERING INTERFACE AND DISCIPLINES

Literature Review

In today's systems, software binds the systems together and causes the desired capability to emerge (Fairley and Willshire 2011b). Software performs more functions than hardware, and those functions can be much more complex (Sheard 2004). Because of the resulting complexity, "our large software systems can no longer be monoliths… tested within known performance limits" (Brownsword et al. 2006). Both fields are essential to the other.

Systems and Software Engineering Interface Issues Can Cause Problems. The US Air Force's Weapon Systems Software Management Guidebook (USAF 2018) describes software problems arising partly because of "ineffective systems engineering interfaces to the software development process." Kasser and Shoshany (2000) blame "massive failures" in complex projects on communication issues between systems and software engineers.

Systems and Software Practices Must Evolve. Vierhauser, Rabiser, and Grünbacher (2014) argue both the traditional systems engineering approach and software engineering approaches need to evolve to address modern systems. Of course, how to evolve is a question receiving much attention.

Attempts To Reconcile. Researchers still frequently cite Maier's seminal paper on reconciling systems and software architecture (2006). Fairley and Willshire

(2011a and 2011b) suggest educating each discipline in the other's knowledge base. Sheard (2014), mapping the two disciplines in a Venn diagram, called for increased collaboration. Giese (2005) and Sheard (2004) looked at how software engineering practices and systems engineering practices, respectively, would need to change for more software-intensive systems in the future.

How Working Together Achieves Integrated Systems and Software Engineering. In 2007, Boehm and Lane published the incremental commitment model to integrate systems engineering, software engineering, and system acquisition. Turner, Pyster, and Pennotti (2009) proposed a "touchpoint" framework for integrating systems and software engineering. This framework notes process faults such as gaps, clashes, and waste, particularly for "interdependent systems" where hardware and software cannot separate, requiring the design to include them in an integrated manner. Boehm et al. (2010) took this further with "architected agile solutions for software-reliant systems," which sounds timely even today. Rosser et al. (2014) further described how to do systems engineering using Agile methods in cross-functional teams.

In 2011, Boehm spoke of "future software engineering opportunities and challenges" requiring significant changes in and integration of both software and systems engineering processes. He focused on generating value and dynamically balancing agility, discipline, and scalability. This same year, Fairley and Willshire (2011b) described education in software engineering that systems engineers need, stating, "Smooth development process integration used in systems engineering and software engineering is a continuing and ongoing challenge."

The Systems Engineering Body of Knowledge (stewarded by INCOSE, the IEEE Computer Society or IEEE-CS, and the Systems Engineering Research Center, and maintained as a wiki) has a detailed section on software. This section includes software engineering in the systems engineering lifecycle, the nature of software, Guide to the Software Engineering Body of Knowledge (SWEBoK) overview, key points a systems engineer must know about software, and software engineering features (Fairley et al. 2019). In contrast, the SWEBoK (most recently published by IEEE-CS as a pdf in 2014) has only two paragraphs about systems engineering and does not connect the definition elsewhere in the book (Bourque and Fairley 2014).

Systems Engineering of Software-Enabled Systems (Fairley 2019) is a new and comprehensive book on the subject. In a discussion with the authors of this paper, Fairley said, "The keys to making my approach work are for hardware, software, and human factor engineers to collaboratively generate system capabilities based on stakeholder requirements, then collaboratively generate the system requirements from the capabilities to develop a Model-Based Engineering (MBE) system architecture model and perhaps the subsystem architectures for large systems, and then incrementally and iteratively replace simulated system elements with real elements as they become available—not as simple as this elevator speech makes it sound, but it is efficient in the overall system development." This paper attempts to make this and other systems-software experience more available than it is today.

INCOSE focus on Systems and Software Engineering

Until 2017, INCOSE had many groups addressing the important software-related issues for systems engineers, but none specifically addressed the system software interface. Hence, interested INCOSE members created the SaSIWG.

Of INCOSE's software-related groups, the Model-Based Systems Engineering (MBSE) Initiative is the largest and most successful. It started in the mid-1990s and has been active for about 25 years. Today, its spin-offs, the MBSE Patterns Working Group and the Model-Based Conceptual Design Working Group, are also active. Other INCOSE working groups with activities related to software include Agile Systems and Systems Engineering, Architecture, Complex Systems, Digital Engineering Information Exchange, Enterprise Systems, Object-Oriented Systems Engineering Method, Resilient Systems, System Safety, Systems of Systems, Tools Integration and Model Lifecycle Management, and Training.

METHOD

This paper provides best practices and problems (or challenges) to inform program (or project) managers, organizational managers, and lead engineers how they should target their work for minimal interface issues moving forward. The method to gather best practices and problems was surveying experts in interviews. The authors analyzed and summarized the key interviewee responses.

Goal. The survey determined what today's software and systems engineers think the systems-software interface problems or challenges are and tried to identify any best practices to help reduce the problems and lead to successful systems.

Pre-interview Work. The SaSIWG brainstormed what questions the survey should include, given the goal. Some SaSIWG members and survey participants helped test the questions and shorten the survey to fit a one-hour interview. The survey included real-world case examples and results.

Interviewees. We conducted individual interviews with 31 self-identified systems and software engineers, most of whom were INCOSE members. SaSIWG participants recommended interviewees should include individuals with significant expertise and representation across industry and geographic regions. Interviewees included systems architects, software architects, and project and program managers with extensive experience in one or both disciplines. They included commercial (aerospace, automotive, communications, microelectronics, medical equipment, technology), defense, and academic domain professionals. Many interviewees had experience in more than one industry. The interviewees had, on average, 25 years of systems engineering experience and 20 years of software engineering experience. In some cases, interviewees held positions involving both systems and software engineering. In such cases, the years of experience included both domains.

Interviews. Each interviewee received a SaSIWG charter copy, the survey purpose, the survey questions, and the 2018 SaSIWG paper. Interviews lasted one to one and a half hours. The interviewee received the interview notes for any further clarifications or corrections. Two authors jointly performed all interviews to reduce variation due to different interviewers. The interviewers asked each interviewee to identify two top priorities from six possible areas needing emphasis in the field.

Analysis. The authors screened comments for relevance, sorted them by category, and summarized them in two lists, (i) "best practices" related to the system-software interface and (ii) key problems and challenges requiring solutions.

Availability. This paper summarizes the best practices, problems, and challenge areas. Traceability to interviews is available. The interviews are anonymous, and the notes from each interview are available upon request.

SURVEY RESULTS

The survey had questions interviewees could answer-and interviewers could analyze-numerically, such as priority areas for improvement. Many other questions elicited stories and guidance, and we detail the guidance, both on best practices, challenges, and problems, below. The executive summary summarizes the survey results for program and project managers and systems and software leads.

Executive Summary

The results show improving the sys-

tem and software interface in today's and tomorrow's complex software-intensive systems requires the organization to perform both systems engineering and software engineering well.

Engineering the system and engineering the system software must connect tightly. Most respondents cited the work should integrate, meaning engineers from both disciplines should be part of the project team from day one.

Model-Based Engineering (MBE) enables collaboration, manages complexity, and improves interfaces. MBE, or digital engineering in the US DoD, is a modeling ecosystem covering MBSE, Model-Based Software Engineering, and other model-based engineering disciplines. MBE recommends the two fields must agree upon the tools, including information sharing, and make them compatible.

This survey identified expertise and cross-training in systems and software engineering as the most crucial areas for improvement.

Best Practices

We gleaned the best practices to improve the systems-software interface from the interviews; this section organizes their conclusions. The first principle is to do good systems engineering (and software engineering, to the extent it includes the practices below). This means developing and holding a vision, doing model-based systems engineering, ensuring interdisciplinary work, focusing the system data and test aspects, and including modularity. The second principle is to accommodate change with rigorous, model-supported change management. The third principle is creating, maintaining, and supporting an effective team.

I. Do "Good" Systems Engineering (and Do "Good" Software Engineering). *Effective interfaces depend on performing systems engineering effectively. Some good systems engineering basic quality attributes include: having a solid and agreed-upon vision, doing model-based engineering, establishing a data architecture, maintaining an interdisciplinary team focused on system success, having a well-designed test plan, building modularity in the design, implementing security, and considering the system's human aspects with support of an effective software engineering practice.*

We can improve the interface between systems and software through using streamlined engineering processes based on standards (15288 and 12207); fewer documents, more efficiency, agile, and model-based engineering. The systems engineering method tailors to the system type, environment, and context. We define interfaces at a high-level upfront in the high-level architecture phase based on the top-level system specification.

Ia. Develop and hold the vision. Systems engineering must visualize what the system will do and ensure subsequent development of both systems and software to implement the vision. This usually requires significant and ongoing collaboration and communication with software engineering and other disciplines. As an example, an excellent top-level specification establishes the overall project objective. Another example comes when implementing systems of systems (SoS), or systems belonging to an SoS: the SoS must have its vision, separate from the constituent system visions. Having a clear vision improves communications and understanding across teams and team members.

Ib. Use Model-Based Engineering. Historically, MBSE increased in scope and importance for over 20 years. Recently, its adoption has benefitted from increasing MBSE-supporting tools. Today, MBSE increasingly coordinates with Model-Based Software Engineering. Model-based system representations improve understanding across interfaces.

Model-Based engineering done right allows multiple engineers in disparate disciplines to co-create the product system, helping engineer various artifacts concurrently. Model-Based engineering also supports methodologies such as Agile and DevOps, which handle increasingly complex products.

Model-Based engineering requires model validation despite its challenges, "All models are wrong, but some are useful"(Wikipedia 2021).

Basing the engineering on correct and consistent models generates valuable artifacts from the models. These include the earliest interfaces, and eventually, Interface Control Documents (ICDs) and other documentation such as requirements and design descriptions. Modeling formalizes agreements, provides traceability, and allows successful changes even when the changing system is complex. Models offer capabilities to customers they have not had before, such as dynamically updating requirements, leveraging machine learning, data analytics, and artificial intelligence. Models can also provide automatic code generation.

The model master database provides a single truth source which forms the interface definition basis.

Various systems modeling languages and approaches, including Systems Modeling Language (SysML), Unified Modeling Language (UML), and Object Process Methodology (OPM), address different needs. Organizations need to select the best approach for their environment and programs.

While less widely known to systems engineers, the industry has successfully used OPM (ISO/PAS 19450). It builds on a minimal universal ontology—object process theorem—to model any system with minimal building blocks and processes with links between the objects and processes. OPM provides a standard model for software and hardware.

Some modeling improving the system-software interface is as follows:

1. Model **requirements** as Use Cases and ensure stakeholder, including customers, agreement.
2. Model the system **architecture**. Include functional, object, and data architecture views, with graphical models for all. Define the "what" before the "how." The architecture and model master database are essential to successful interface definition.
3. Begin the top-level **system specification**, which documents the requirements for the system or product, in a model-based description and obtain critical stakeholder buy-in and agreement before finalizing it.
4. Select the right tools based on the problem. In the systems world, no tool is "one size fits all." Using state-of-the-art and connected **tools** enables better interface development. Using the same tool for systems and software is helpful. Tools with structures and schema that are very strict on controlling the Interface Requirements Specification (IRS) and the Interface Data Specification (IDS) are beneficial. Strict controls implement the interfaces through auto code generation tools. The tools must support the users. Do not over tool; provide tool support for users.
5. Model **interfaces**, upfront and as they change. A system model database provides the single truth source for interface definition. Ensure all relevant stakeholders' agreement on the interfaces. Use models to demonstrate traceability. Generate important documents (requirements and ICDs) from the models. Interfaces will change; use rigorous change management supported by the models track changes (see below). Treat additional model inputs from the components as either conforming with or changing those rigorously controlled interface specifications. Some organizations replace interface documents with models. Some companies use the model database and escalation processes to manage interface changes.

6. **ICDs**, or the model-based equivalent, describe the system presenting the interface, the inputs and outputs passing through the interface (typically energy, force, material, or information) including the optical/visual, the behavior at the interface (described by interactions) and the access system (medium) supporting the interactions. More general requirements include agreeing on how subsystems talk to each other, error conditions, and actions upon detecting an unexpected input or event.
7. Plan and **model the verification and validation** needed to prove components and the interfaces, ensuring the larger level system's correctness. These can be part of the modeling effort.
8. Use **simulations** and executable models to capture, first desired and eventually actual, behavior and algorithms showing the variable relationships.
9. Use **prototypes**. System and software define this word differently (Fairley 2019; Sheard 2016). Systems engineering means a model with functionality in place but immature technology. Software means a model with specific functionalities in place but with other functions simulated ("stubbed"). Prototyping either type, along with simulation before architecture formalization, helps develop the system and software. Throughout the lifecycle, prototyping also resolves open technology questions and demonstrates functionality to users.

Ic. Ensure interdisciplinary focus on system success. Systems, software, and hardware must agree on all major decisions, and other disciplines must agree with whatever involves them. In general, systems engineering's history of integrating disciplines needs to continue, increasing its emphasis on involving software engineers and architects. Strong communications are essential. For example, software engineers must communicate with systems engineers if requirements are ambiguous.

The teams use common or connected processes, connected tools, views, taxonomies, and review metrics throughout the lifecycle and across systems, hardware, and software teams. In some software-intensive systems, no interface exists between the system and the software because the two are not separate. There is a systems perspective, and the software is integral to the system. There is extensive prototyping and simulation to develop the system.

Regardless of the organizational structure or job titles, the person or team performing the systems engineering is responsible for system success.

It is vital to use root cause analysis and analytics to understand the problem area's root cause, including interface defects, to identify actions for continuous improvement.

Id. Address data as a whole. Address data with a system-wide focus (Digital Thread—data interface and traceability from disparate engineering models). Data has always been there; more so, product and process data in current complex systems are more numerous and complex than ever. Hence, data should at least have a taxonomy, architecture, plan, and single truth source.

Ie. Address test as a whole. Software testing terminology and differs in some ways from testing and verifying systems. Testing should focus on models. In general, software teams frequently test every Agile cycle (if they use Agile processes, which most do). Systems engineers need to understand this to extend it to systems engineering, where appropriate, and guide it to help meet the system vision. Developing the test harness while building the system is necessary to support continuous testing.

If. Implement modularity. Managing the development of increasingly complex products requires modularity. It supports developing some basic functionality first and adding complexity later, such as with Agile and DevOps.

Ig. Implement security from the beginning**.** Security challenges today focus on cybersecurity. Cybersecurity challenges change with every zero-day exploit, which software engineers will most likely know about before systems engineers. The initial design team should involve security engineers, and they should approve all changes.

Ih. Do good software engineering. It is helpful to define systems and software engineer roles. IEC 62304, Software Development Process (IEC 2006) helps with this. Requirements flow to software engineering for implementation. Good software engineering practice provides a focus on the system, rather than just on code and objects. The system can evolve as we incrementally add software when using a subsystem approach. If software development uses agile practices, communication is a critical process component. Systems and software groups agree on interfaces before software development for the essential system parts. There is a top-down approach (plan-driven approach, requirements, architecture, protocols, interfaces, and interactions) for systems and interfaces and a bottom-up approach for development teams, who perform continuous development with flexibility. Successful agile practices adequately map system elements to requirements. The Scaled Agile Framework (Leffingwell 2018) often applies to large systems. Complex projects with more software use a DevOps approach to capture user requirements and improve software quality. This impacts the system and software interface definition. The DevOps approach (Bass, Weber, and Zhu 2015) builds modular pieces and adds complexity later in layers. DevOps embraces developing what you create as you build it. The trade space between hardware and software can evolve. Other key aspects include continuous integration, prototyping, and commercial off-the-shelf subsystems.

Ij. Build systems with consideration to humans in systems. Include users in determining requirements through scenarios, in major reviews, and, at a minimum, operational testing.

II. Evolve Systems Engineering Practices. Manage change robustly and effectively.

IIa. Be adaptable to process change and improvement. Processes bringing real-time feedback and product or system evolution (Agile and DevOps) necessitate systems engineering processes built to adapt to change. Systems engineers use spiral or Agile lifecycle management processes, when appropriate, instead of the single-pass waterfall. Customer engagement is essential, regardless of the lifecycle model.

Evolve processes to facilitate concurrent engineering enabled by digital engineering and the digital thread. Concurrency strengthens the worldview of the team, the organizational structure of people, and improves interfaces.

IIb. Manage changes—update interfaces. Perform rigorous change management across the program (any change might affect interfaces) to ensure the changing system-software interfaces become visible during the change approval process and be ready to update interface specifications.

IIc. Enable systems engineering evolution. Plan for several systems engineering aspects to evolve. These include MBE, artificial intelligence (AI) and machine learning use, and approaches for understanding complexity and emergent behaviors.

Processes and approaches evolve to support better system evolution, such as re-evaluating previous decisions when technology or needs change while considering matching the solutions to criteria and the criteria importance. For example, the trade space between system hardware and software may change. To ensure flexibility, the system engineer leaves options open as to where to perform functions as long as possible as the system evolves.

System and software engineering processes evolve to use the same system and software lifecycle process and unified thinking.

Table 1. Top problem areas identified by interviewees

Risk Area or Opportunity to Improve	# times top*
1. **Expertise**. Systems team lacking software expertise or software team lacking systems expertise or other expertise issues	16
2. **Interface definition or spec**. Lacking interface definition including data, specifications	14
3. **Leadership**. Organizational or management issues	12
4. **Process**. Lacking agreement on critical processes, including process problems in specific lifecycle phases from concept through maintenance or problems in language impacting interface and system performance	11
5. **Environment & System Type & Other**. Problems related to certain system types—complex, emergent behavior, or distributed control (Note: Four respondents indicated problems with complex systems such as autonomous systems, emergent behaviors, and lack of tools and processes. One respondent cited problems with test architecture.)	5
6. **Technology and Tools**. Lacking modeling tools or other tools or incompatibility among tools; technology gaps	4

*# times ranked in top two problem areas by an interviewee

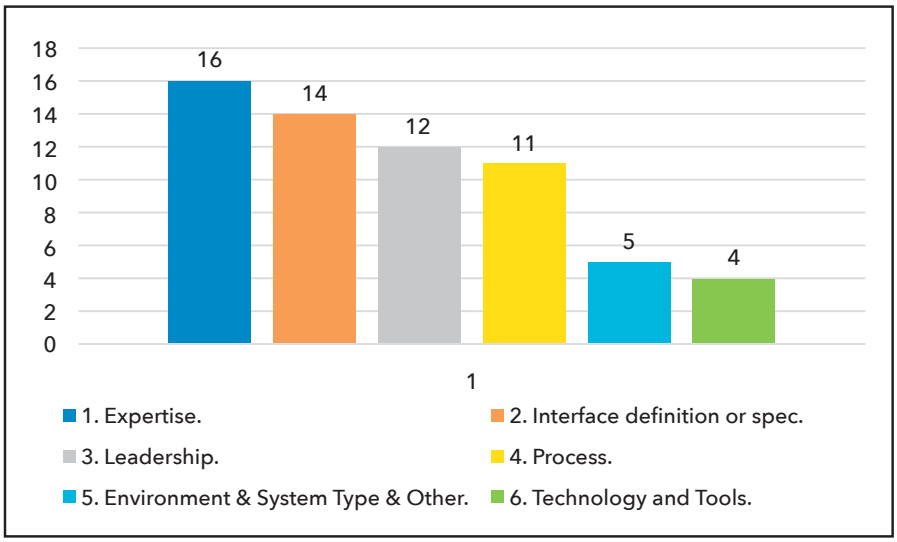


Figure 1. Top problem areas

III. Ensure Teams Have Expertise for Increasingly Software-Intensive Systems. Effective teams include people who have the right expertise and behaviors, who trained in what they do not know and in what new capabilities and technologies are available, and who can communicate appropriately with terminology interpretation awareness. Effective teams have organizational leadership support.

IIIa. Assure team expertise and effective communications. The system-level team includes people with system and software architecting and engineering expertise. Ideally, architects and requirements elicitors have both systems and software expertise. The team includes both software and hardware engineers.

The team comprises people who understand technical limits, including physical. The team understands the problem to understand what software to build. The team also includes people with expertise in modeling, modeling languages, and modeling and simulation tools. Both systems engineers and software engineers must know how to create and implement Use Cases, which help bridge disciplines and customers (see Fairley 2019, on skill importance in Model and Simulation-Based Systems Engineering to build systems architecture models).

Communication is key. Both system and software team members should understand terminology from both disciplines (and move toward a shared vocabulary). Team members treat each other respectfully, and the team addresses cultural barriers and office separation problems to improve effectiveness.

Some teams on software-intensive systems fully integrate systems and software people to improve effectiveness.

IIIb. Assure leadership supports best practices and understands the software's importance. Leadership supports the proper discipline integration from day one, without silos or organizational interfaces. Leadership builds a supportive culture and understands the importance of collecting needed skills. Systems engineering and software engineering are not the same. Both systems engineers and software engineers bring essential insights throughout the project. Leadership recognizes system-team expertise in software is a crucial differentiator for their products.

Leadership also recognizes processes will change to improve the interface, and in the future, there will probably be more Agile and less "big design up front" or "waterfall" process. In an effective Agile team, there is no "us" and "them." An Agile environment demands the best people and a good environment. An Agile team works together on solutions, and leadership participates rather than commands and controls.

Priority Areas for Improvement

Each interviewee identified his/her top two priority areas for improvement. We used the categories identified in the first SaSIWG INCOSE paper for the survey. Table 1 shows how frequently interviewees identified each Risk or Opportunity area. Figure 1 shows these same numbers in relation to each other.

Key Challenges and Problems

This section summarizes the key challenges and problems answers.

We mapped the challenges and problems

into the "best practices" categories above and tagged them with the applicable roman numerals and letters.

Expertise. (IIIa) Lacking expertise in both software and systems is a significant issue. System engineers need some software engineering knowledge, and vice versa. If we resolve this issue, many other issues will resolve as a result.

With digital engineering and software-intensive systems, system engineers must have good software insight. Software engineers must also understand the system. The systems engineering team may have dated experience. System engineers and software engineers must stay up to date through education and knowledge-sharing. They do not need to be experts. The software team needs systems thinking knowledge and tools. Software engineers need to obtain, acquire, and assimilate the need for higher-level system engineering, such as why and how systems engineering is a discipline they need to interface with and why software engineering needs system knowledge.

Shared Objectives; Interdisciplinary Teamwork. (Ia, Ic, IIa, IIc, IIIa, IIIc) Alignment issues on objectives between organizations cause systems problems. The project must involve systems engineers from the start in a collaborative process. Sometimes, the project brings in systems engineers too late. Conversely, in the semiconductor and automotive industry, sometimes the systems architect specifies the system (and its architecture) without consulting the software architect. In the past, the software was the system in domains/companies with software-intensive projects. Today, software resides in a more complex environment and needs to be an integral subsystem. The system process also needs to integrate the software DevOps approach. Functional organizations create barriers to developing a systems focus.

Cultural Alignment; Communications; Relationships. (Ic, IIa, IIc, IIIa, IIIb) Systems and software engineers need to understand one another—from understanding each other's worldview at a high level to communicating with each other's tools at the detailed level. We must develop practices to support the dual need for "build to last" and "adaptability."

We need to understand and resolve interpersonal communication barriers, including language and behavioral differences (Kasser and Shoshany 2000). All stakeholders need common (or agreed to) ontologies, terminology, and semantics across systems, software, and hardware. Often friction appears between software engineers who force system engineers "out of the clouds" and system engineers who push software engineers to understand the world is bigger than the computer screen. We call engineering leaders in the defense industry system engineers, though the individuals may lack the necessary tools and skills. These positions require software architecture skills as well. Today, systems engineering and software engineering practices do not support joint discovery and requirement evolution as necessary and natural.

The trade space between software and hardware frustrates engineers in both fields. It is important to allow for requirements and the trade space for software and hardware to adjust as a program evolves, and more is known about the environment as it changes and the system develops. It is important to understand systems and software competencies and differences, particularly given the blurring and blending trends between systems and software (see Fairley 2019, page 39 regarding methods and common attributes).

Lacking Good Systems and Software Engineering. (I, Ic, Ih, IIa) Systems and software engineers expressed a need for more agile processes to avoid commitment to a big upfront design. Software often makes up what was forgotten in the system, too late in the lifecycle, in band-aid mode. The systems engineering team did not understand the problem, missed something, or could not discern the customer's wants. Software engineers who do not realize the big picture cause problems. For example, we cannot add safety at the product development process' end. Creating a big picture view helps even when software engineers do not recognize the importance. Invite customers to talk to software engineers (who are often so focused on deliverables they do not take time to talk to users). We need customer involvement and focus in the process, including design for usability/human factors. In some industries (consumer appliances or medical products), systems engineering practices are new and not well established.

Education; Engineering Disciplines. (IIa, Ic, I) Some interviewees indicated systems engineers rarely have formal education in system engineering. The best systems engineers have broad engineering experience. The systems engineering and the software engineering disciplines are not identical. More than one interviewee said systems and software engineering should be two specializations within one educational department. Software engineers need to understand the greater lifecycle. Systems engineers need to understand the fundamental system building blocks. Education programs must keep pace with software-based Agile technologies, techniques, and tools.

Process Challenges. (I, Ib, Ib1, Ie, Ih, IIc) Complexity is increasing the need to improve systems engineering processes with systems thinking and impact analysis. System thinkers need to test what they do before they do it and understand the emergent behaviors. Modeling and simulation are necessary, but it may not be possible to predict emergent behaviors in complex systems. Modern systems engineering practices do not fully address modern system challenges and opportunities, such as complexity, AI, deep learning, and emergent behaviors.

Examples:

Implantable medical devices have new input sources—a user can modify the system's behavior rather than requiring a doctor, or else other emergent behaviors bring more complexity and the need for a SoS approach.

Autonomous vehicles need new approaches: maybe requiring a software patch before finding the problem's root cause (fuel gauge sensor).

Software validation must validate all lines of code. Models require validation, and models may need refactoring to realize a digital thread. Can we use advanced technology (AI and machine learning) to validate models?

Risk management is important. Risk management is often weak or dependent on another technology not at an acceptable readiness level.

It is critical to use a system development lifecycle process integrating systems engineering, software engineering, and program/project management. One common issue is teams from the different domains diverge after the cross-domain Use Case step, developing artifacts no longer familiar to stakeholders who provided the original needs. Cross-domain, in this instance, means developing specific Use Cases to elicit an understanding of disparate project domains (engineering or business)

Interface Definition. (Ib, Ib2, IIb) After analyzing 50 companies' systems engineering competencies, we identified interface management as the weakest systems engineering capability (Source: Mark Sampson, Siemens). Model-based representation improves understanding across interfaces. Understanding across interfaces is essential with software-system interfaces. The INCOSE MBSE Patterns Working Group (in collaboration with NASA, JPL, and OMG) worked on sharable patterns improving the interface representation.

Sometimes we do no define interfaces well because of an incomplete understanding of the enterprise architecture and data

model. Problems can also arise with a lack of planning/design at the front end, lack of information available from customers or suppliers, or too many items still undetermined in the Technical Requirements Document or the Interface Control Document.

Simulation. (Ib8) Systems and software engineers have expressed concerns about gaps in the capabilities and tools to do system performance simulations. We need to progress beyond static documentation and structural models to executable models to ensure interfaces and system performance. Improving the interface requires MBSE evolution (integrating requirements into modeling, using models to simulate systems, or models to determine performance). SysML models are currently static. The SysML language and tools must mature to provide a native dynamic simulation. MATLAB can do some simulations, but this happens through an external interface integrated into the SysML model utilizing parametric diagrams. Systems engineers would like to push a button in the model to validate the system's performance.

Interface Security. (Ib, Ig, Ih) From the cybersecurity viewpoint, there are significant, growing trust problems in the interfaces. Systems have to be flexible and change over time. It is continually necessary to define what input data is and ensure trust with input and output data. As interconnected systems change, we must ascertain if we can still trust the input and output. Poorly done software (hacking code or sometimes using the "Agile" excuse) is a huge problem. As technology changes, this presents additional challenges for cybersecurity. Software languages and libraries change so quickly the project has insufficient time to check for fit-for-purpose and security.

Data Volume. (Ib2, IIc) Best practices lack handling data volume and complexity, millions of data points. Engineers and operators can become overwhelmed with data of unknown importance. Some interviewees saw a need for better data management infrastructure. We currently do not have well-established interface management for complex systems (for aircraft with 1.2 million interfaces, captured as individual things and managed as individual things). Some interviewees felt we could use databases better to manage this data complexity. When an interface changes, teams should notify the right people to determine the impacts and control integration. Interfaces can then build on architectural decisions.

Other Modeling Challenges. (Ib, IIc) Using model-based engineering to establish requirements makes it imperative to improve practices to keep the models fresh. This area is ripe for research in including accountability and ensuring more rigor during product development. Systems interface requirements need to update dynamically based on data collection, leveraging advanced analytic practices, and machine learning and AI. Ambiguity exists between SysML and UML (both are drawing languages; while SysML evolved from UML originally, both have evolved somewhat independently). The workaround uses data architecture (data schema/ontology) and simulation tools to bridge the ambiguity. A common ontology, semantics, and terminology would be helpful. For example, in the automotive domain, hardware/mechanical and software have somewhat different terminologies.

Agile and Concurrent Engineering Expertise. (Ih, IIa, IIc, IIIa, IIIb) Interviewees noted gaps in understanding practices for engineering Agile systems and software and in performing concurrent engineering. For example, in Agile, there are always unknowns throughout the process, but design efforts always proceed, assuming the unknowns will successfully resolve later. The team may discover something they planned cannot be done (though using the rolling wave planning process helps).

Systems of Systems. (Ia, IIc) Interviewees noted a need for unambiguous systems or systems of systems (SoS) objectives and metrics for success. Is there a way to indicate when the parts should sub-optimize to optimize the whole? How can we create systems to work for themselves yet also make the SoS work together? System owner motivations and incentives in a SoS often do not align. (This is an area for additional research and development.)

Discussion and Future Work

We consider this survey of 31 interviewees complete, although the SaSIWG is always interested in hearing from others and will incorporate additional answers into ongoing work. Future work will entail more discussions within the SaSIWG and other INCOSE working groups to clarify and communicate best practices. Fairley (2019) has added considerably to the best practice knowledge base derived from the interviews.

The SaSIWG also plans to address problem areas. As a first step toward approaching the expertise key challenge, the SaSIWG conducted a "Book Club" (professional development) webinar series discussing Fairley (2019) and has recorded it for future systems and software engineer education.

CONCLUSION

The INCOSE member and other systems and software engineer survey captured numerous helpful suggestions for organizations to adopt as best practices. These include leveraging existing best practices in systems and software engineering, evolving systems engineering practices (agile, MBE/MBSE, concurrent engineering, and AI/Machine Learning), and assuring teams have the expertise for increasingly software-intensive systems.

The survey also identified the most critical challenges for engineering leaders and systems and software engineering leadership. The top challenge area was expertise—lacking software engineering skills among systems engineers and vice versa. Ranked second was the lacking interface definition, including data and specifications. We must address these challenge areas to improve software-intensive system delivery.

While important, recommendations and helpful suggestions alone may not sufficiently address the pain points. Additional discussions related to implementing best practices, strengthening expertise, and evolving systems engineering practices are vital. Implementing additional education and cross-training between systems and software engineering, MBE/MBSE methodologies, and concepts such as machine learning, data science, and artificial intelligence have provided some tools to address the pain points. Together, these can augment engineering product lifecycle processes while enforcing discipline within rigorous and judiciously chosen processes.

SaSIWG will continue to share the survey findings to address critical problems and challenges, including education and systems and software engineer training in novel ways. ■

REFERENCES

- Bass, L., I. Weber, and L. Zhu. 2015. Devops: A Software Architect's Perspective. Boston, US-MA: Addison-Wesley Professional.
- Boehm, B. 2011. "Some Future Software Engineering Opportunities and Challenges." In The Future of Software Engineering, edited by S. Nanz, 1-32. Berlin, DE: Springer-Verlag Berlin Heidelberg.
- Boehm, B., and J. A. Lane. 2007. "Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering." CrossTalk 20 (10): 4-9.
- Boehm, B., J. A. Lane, S. Koolmanojwong, and R. Turner. 2010. "Architected Agile Solutions for Software-Reliant Systems." In Agile Software Development, edited by T. Dingsøyr, T. Dybå, and N. B. Moe, 165-184. Berlin, DE: Springer, Berlin, Heidelberg.

- Bourque, P., and R. E. Fairley, eds. 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0. Washington, US-DC: IEEE Computer Society Press.
- Bramer, J. 2017. "Integration of Software and Systems Engineering." Paper presented at the Annual International Workshop of INCOSE, Torrance, US-CA, 28-31 January.
- Brownsword, L., D. Fisher, E. Morris, J. Smith, and P. Kirwan. 2006. "System-of-Systems Navigator: An Approach for Managing System-of-Systems Interoperability." Technical Note, Software Engineering Institute. DOI: 10.1184/R1/6584582.v1.
- Fairley, R. E. 2019. Systems Engineering of Software-Enabled Systems. Hoboken, US-NJ: Wiley.
- Fairley, R. E., T. Hilburn, R. Madachy, and A. Squires. 2019. "Systems Engineering and Software Engineering." Guide to the Systems Engineering Body of Knowledge (SEBoK). https://sebokwiki.org/wiki/ System_Engineering_and_Software_Engineering.
- Fairley, R. E., and M. J. Willshire. 2011a. "Teaching Systems Engineering to Software Engineering Students." Paper present at the 24th IEEE-CS Conference on Software Engineering Education and Training, Honolulu, US-HI, 22-24 May.
- Fairley, R. E., and M. J. Willshire. 2011b. "Teaching software engineering to undergraduate systems engineering students." Paper presented at the 2011 American Society for Engineering Education (ASEE) Annual Conference and Exposition, Vancouver, CA-BC, 26-29 June.
- Giese, H. 2005-2006. "Software Engineering for Software-Intensive Systems: I. Introduction." Software Engineering Group Winter semester, University of Paderborn (Paderborn, DE).
- IEC (International Electrotechnical Commission). 2006. IEC 62304:2006. Medical Device Software-Software Lifecycle Processes. London, GB: IEC.
- ISO (International Organization for Standardization)/IEC (International Electrotechnical Commission)/IEEE (Institute of Electrical and Electronics Engineers). 2015. ISO/IEC/IEEE 15288:2015. Systems and Software Engineering—System Lifecycle Processes. Geneva, CH: ISO/IEC/IEE.
- ———. 2017. ISO/IEC/IEE 12207:2017. Systems and Software Engineering—Software Lifecycle Processes. Geneva, CH: ISO/IEC/IEE.
- Kasser, J., and S. Shoshany. 2000. "Systems Engineers are from Mars, Software Engineers are from Venus." Paper presented at the 13th International Conference on Software and Systems Engineering and Their Applications. Paris, FR, 5-8 December.
- Leffingwell, D. 2018. SAFe 4.5 Reference Guide: Scaled Agile Framework for Lean Enterprises. Boston, US-MA: Addison-Wesley Professional.
- Maier, M. W. 2006. "System and Software Architecture Reconciliation." Systems Engineering 9 (2): 146-159.
- Rosser, L., P. Marbach, D. Lempia, and G. Osvalds. 2014. "Systems Engineering for Software Intensive Projects Using Agile Methods." Paper presented at the 24th Annual International Symposium of INCOSE, Las Vegas, US-NV, 30 June-3 July.
- Sheard, S. A. 2004. "Adapting Systems Engineering for Software-Intensive Systems." Paper presented at the 14th Annual International Symposium of INCOSE, Toulouse, FR, 20-24 June.
- ———. 2014. "Needed: Improved Collaboration between Software and Systems Engineering." Software Engineering Institute Blog, 19 May. https://insights.sei.cmu.edu/sei_blog /2014/05/needed-improved-collaboration-between-software-and-systems-engineering.html.
- ———. 2016. "What Do Systems Engineers Need to Know about Software?" Paper presented at the National Defense Industrial Association Systems Engineering Conference, Springfield, US-VA, 24-27 October.
- Sheard, S. A., R. Creel, J. Cadigan, J. Marvin, L. Chim, and M. Pafford. 2018. "INCOSE Working Group Addresses System and Software Interfaces." Paper presented at the 28th Annual International Symposium of INCOSE, Washington, US-DC, 7-12 July.
- Sheard, S. A., M. Pafford, and M. Phillips. 2019. "Systems Engineering-Software Engineering Interface for Cyber-Physical Systems." Paper presented at the 29th Annual International Symposium of INCOSE, Orlando, US-FL, 20-25 July.
- Systems and Software Interface Working Group (SaSIWG). 2017. "Approved Charter." Charter, San Diego, US-CA: INCOSE.
- Turner, R., A. Pyster, and M. Pennotti. 2009. "Developing and Validating a Framework for Integrating Systems and Software Engineering." Paper presented at the 3rd Annual Systems Conference of IEEE, Vancouver, CA-BC, 23-26 March.
- United States Air Force (USAF). 2018. Weapon Systems Software Management Guidebook. Washington, US-DC: USAF.
- Vierhauser, M., R. Rabiser, and P. Grünbacher. 2014. "A Case Study on Testing, Commissioning, and Operation Of Very-Large-Scale Software Systems." Paper presented at the 36th International Conference on Software Engineering, Hyderabad, ID, 31 May-7 June.
- Wikipedia. 2021. "All Models are Wrong." https://en.wikipedia.org/wiki/All_models_are_wrong.

ABOUT THE AUTHORS

Sally Muscarella spent 21 years at AT&T and Bell Labs. She has held leadership positions in product management and systems engineering for software-intensive operations support systems, network management, and customer service operations. Sally joined Stevens Institute of Technology in 2013 to lead corporate and government education programs for the School of Systems and Enterprises, holding the position until 2020. She earned a B.A. from Simmons College and an M.B.A. from The Wharton School, University of Pennsylvania. She joined INCOSE and has been active since 2015. Sally enjoys traveling, reading, volunteering, skiing, and playing tennis.

Macaulay Osaisai, a system engineer developing sensor-related systems at L3Harris Technologies, has many years of commercial sector systems engineering experience. Previously, he developed autonomous sensor systems for seismic and geophysical applications. His expertise areas are systems architecture, systems modeling, embedded hardware and software systems, and low SWaP (Size-Weight-and-Power) systems. As an INCOSE member and a senior IEEE member, he is an MBSE evangelist, developing MBSE processes, procedures, and training materials. Macaulay conducts systems and MBSE training classes, consulting, coaching, and mentoring, and he enjoys music, tennis, and scuba diving.

Sarah Sheard is an INCOSE Fellow, CSEP, and Founder's Award winner. An INCOSE member since 1992, she chaired INCOSE's SaSIWG from 2017-2021. Her many systems engineering publications include three INCOSE "Best Papers." When she retired in 2019, Dr. Sheard was a systems and software engineering researcher and consultant at CMU's Software Engineering Institute. Previously, she worked at the Systems and Software Consortium, at Loral/IBM Federal Systems, and Hughes Aircraft Company. In 2012, she earned her Ph.D. from Stevens Institute of Technology in systems engineering, focusing on system development complexity. Now, she postponed international travel to learn folk dancing by Zoom with her husband.

"Book Club" Guides A Working Group to Create INCOSE System-Software Interface Products

Sarah Sheard, sarah.sheard@gmail.com; **Mickael Bouyaud,** mickael.bouyaud@ingenico.com; **Macaulay Osaisai,** Macaulay.Osaisai@L3Harris.com; **Jeannine Siviy,** jeannine.siviy@yahoo.com; and **Ken Nidiffer,** knidiffe@gmu.edu

■ **ABSTRACT**
Many software-dominant organizations ignore systems engineering completely. A recent popular book described one such organization in a fable format. The INCOSE Systems and Software Interface Working Group chose to read this book as a weekly professional development seminar to investigate where systems engineering should fit in software-dominant organizations of the type that do not currently involve INCOSE's systems engineers. This seminar culminated in an "author day," where the book's author responded to our questions and discussed our potential role. The book club activities produced several drafts we can turn into products to help INCOSE systems engineers better understand software, and systems engineers' potential role in such organizations, to help improve software-intensive system success.

THE SELECTED BOOK

INCOSE's Systems and Software Interface Working Group (SaSIWG) wanted to explore the role systems engineering would have in future software-dominant organizations. While some organizations today, notably defense and safety-critical systems industries, have a history of systems and software working together, others, notably technologically explosive and disruptive commercial businesses, do not. Because *The Unicorn Project*, by Gene Kim, describes the problems this type of business faces, the SaSIWG selected this as our second professional development or "book club" project. We also selected it because the industry described in the book does not include systems engineers and is unfamiliar to many INCOSE members.

DISASTER TO HOPE THROUGH REBELLION

At the beginning of *The Unicorn Project*, silos dominated the fictional organization, as in many real organizations. There was no consistent software development environment. The integration test environment had no architecture.

Management interfered initially, and engineers could not talk to each other without obtaining management's approval, though several enlightened managers discovered and admitted this throughout the story. Managers repeatedly outsourced Information Technology (IT) to reduce cost, then re-integrated because outsourcing made IT too unresponsive. Every change caused significant disruptions. Project managers functioned as paper pushers who primarily created more dependencies, adding wait time and complexity.

No one considered the whole system, only their specific portion, because no one understood more than their portion. There were no systems engineers, and no one understood the need for a system viewpoint. Teams even considered architects harmful, people who just sit on committees and make people complete forms.

Managers without software development backgrounds (such as the character Sarah Moulton) had a poor software development understanding, had the wrong attitude (blame and punish), and made the wrong decisions (to sell off the company for parts).

The book's hero, Maxine, gradually worked to understand the situation, and in doing so, found like-minded people who helped her implement a new working method. They had begun operating in hero mode, with the motto, "Breaking rules is the only way to get things done." Through several pilot projects and reorganizations, Maxine increased their higher management support, gave customer delivery speed more attention, and broke down archaic anti-patterns preventing successful work. Management at higher levels began to

Table 1. Terminology table

Term	Definition	Example or Opposite	Notes
Dev environment	Tools and procedures for developing, testing, and debugging an application or program.	IDE: Integrated development environment provides developers with standard UI (user interface).	Techopedia says a dev environment normally has three server tiers, called development (developer), staging (testing production for reliability), and production (actual run environment).
Docker	Products deliver software in packages called containers. Containers bundle their software, libraries, and configuration files; isolate from other containers; and communicate through well-defined channels. A single operating system kernel runs containers and therefore uses minimal resources.		Docker is a company, released as open-source in 2013. There is software, namely a Daemon (a persistent process managing Docker containers, handling container objects, and listening for requests), Objects, and Registries.
ERP System	Enterprise Resource Planning (ERP) systems integrate the software needed to run a system, including planning, inventory purchasing, sales, marketing, finance, and HR.	Oracle, SAP, and Microsoft.	ERP systems evolve often and can be vast and problematic.
GitHub	Web hosting for software development management service. Rely on GIT as versioning tool.		GitHub is also a company developing software and providing services.
HIPPO	Highest paid person's opinion.		Leadership snark.
JIRA	Ticket-based online tool to track bugs, manage incidents and projects.		Tool often used in software projects since it implements agile, scrum, and Kanban.
ODB-II	Onboard Diagnostic Port-2.	Port to an electronic system to tell how the computers on a car are working.	Automotive part component.
POS Registers	Point of sale registers.	Cash registers for the stores selling auto parts in the company.	Registers like cash registers, not like computer registries.

support their effort despite many threats to their jobs along the way.

A SYSTEMS ENGINEERING CONCLUSION

Ultimately, the excellent material success of her team's approach following their "sensei's" advice (the "sensei" transparently representing the book author) led to victory over all obstacles. Maxine received a promotion to "distinguished engineer." The SaSI-WG's 2021 paper (Sheard et al., 2021) argues this position is very close to a senior systems engineer. The working group thinks if the company had started with a position, group, or perspective like this, it never would have fallen into trouble in the first place.

WORKING GROUP PRODUCTS BEGUN DURING BOOK CLUB

- ***Working Pattern for a Systems Engineer in Software World***

  - Assess whatever topic/issue is at hand, whether the big picture as a whole or an immediate problem.

  This might be a quick conversation or a large meeting with "techniques" (post-its or multi-voting) in play. It all depends on how much context the systems engineer already has, how much bias needs weeding out, and how much consensus the team needs to build.

  - Draw pictures to understand it—box the scope, note the interfaces (where everything breaks), map the interrelationships, and clarify the purpose.

  To explain how things work, systems dynamics diagrams and causal loops work well, but systems engineers should draw anything needed to describe functions and data flow depending on what will convey the point. Some pictures may be formal and persist through the project, but often they are messy, temporary, and "fit for use," supporting or prompting a conversation to achieve "aha" moments.

  - Find the pattern, relate it to patterns seen elsewhere, determine the solution, plan it, and move on.

  Patterns can come from principles as well as heuristics and repeated scenarios in other domains. Systems engineers cross-pollinate constantly. This involves "playing it forward"—not only looking at today's static condition but thinking through future circumstances to ensure robustness, resilience, and anti-fragility. A solid statistics, experiment design, or design engineering background really helps.

  - Spend time trying to explain the above to others who are not systems oriented.

Managers sometimes tell systems engineers why the "new" design (or other) methods are somehow superior tools in the toolkit, as if we did not understand, even when we know their common roots and tool evolution. We may have to convince them we do know.

- Be happy when everyone starts rowing the boat together; persist when that does not happen.

Know when to let go but be persistent when it is important. If it is not safety or money, sometimes letting something go "splat" is the way other people "get it"—it just takes forever.

- ***Terminology collection***

A confusing part for a systems engineer in a software-dominant organization is the unfamiliar jargon (words, abbreviations, and acronyms) used without explanation. While reading the book, we started writing down many software-specific terms. Because many systems engineers experience this in software-dominant organizations, the group realized INCOSE members might appreciate a top-level or systems-view explanation that does not require searching through multiple bottoms-up software-type explanations to learn what they mean in a systems engineering context.

We have created a terminology table to turn into a database. Table 1 shows a few terms and a few columns from this table. We need to consider how to provide this information to members. A briefer explanation would be desirable immediately upon first hearing the term, rather than a longer definition later. We also need to consider how we would recommend systems engineers learn the term the "second time," when there is time to digest what the term means in a broader software engineering context.

We are considering turning the table into an online accessible database for INCOSE members. The question is how it would be most beneficial. We are also considering how to turn this effort into INCOSE-sponsored software engineering education for systems engineers, whether online asynchronous, lunch-and-learn, or another learning method.

- ***Drawings of Software Development and Product Delivery Process***

Software development processes also puzzle systems engineers. Our 2021 symposium paper discusses this in more detail than this *INSIGHT* article can, but we show one figure discussing how software developers modify code. A developer starts a new version by cloning the repository

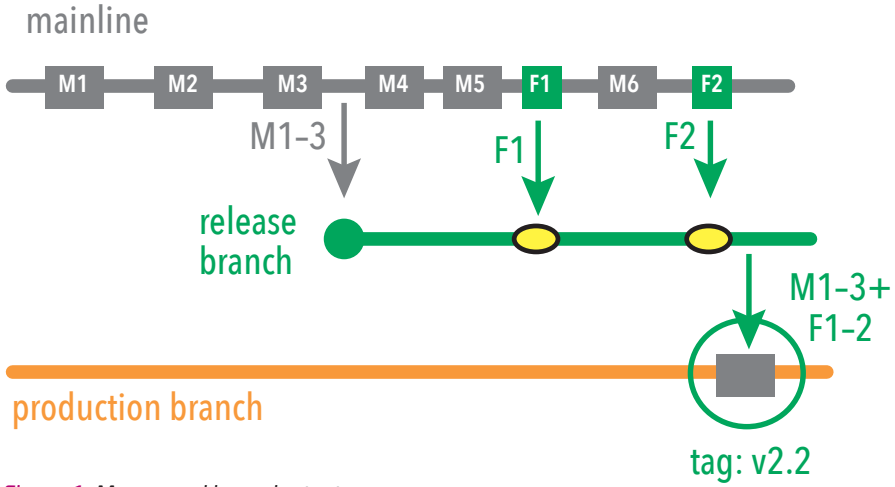


Figure 1. Merge and branch strategy

stored code, issued from a baseline, to his or her computer. After making changes, he or she "pushes" the modified code (using the checkout/check-in process, also called the commit process). Peers can then review this new code and do unit tests. Once approved, they "merge" this new code into the main branch, creating a new baseline. This process is part of a merge and branch strategy, illustrated in Figure 1. Today, these processes, labeled continuous integration, usually use automated tools, such as Maven, Nexus, or others, created specifically for this purpose.

- ***Operational Process Flow Model***

We also made an operational process flow model as the first step toward an executable model to simulate the original organization's process chaos and show how the improved organization works better. Again, we refer the reader to the 2021 symposium paper. The objective is to emphasize the need for a systems engineering role, develop a holistic view enabling early activity gap and redundancy identification, and understand activity and interface dependencies. The systems engineering role is like a symphony orchestra conductor: a thorough understanding of how different instrument sounds combine enables the conductor to produce excellent music and guide the orchestra musicians.

- ***Complex Systems Model***

The Working Group developed models describing organizational approaches to accomplishing work and their limitations, including the monolithic approach, the choreography approach, and an autonomy approach more consistent with complex systems principles. We introduced these to show INCOSE why systems of systems work better with focused autonomy and what conditions they require. Brian White, a former Complex Systems Working Group chair, has written a body of work on this topic (White 2016).

- ***Change Agent Model***

Previous work by Working Group members, including Sheard and Siviy, suggest systems engineers can be effective change agents (McKinney et al. 2015).

NEXT STEPS

The working group will consider which steps to take next to bring these draft and potential products forward to be most beneficial to INCOSE and the Systems Engineering community. To help, please contact the WG Chair, Nick Guertin. ■

REFERENCES

- Sheard, S., M. Bouyaud, M. Osaisai, J. Siviy, and K. Nidiffer. 2021. "A Guide for Systems Engineers to Finding Your Role in 21st Century Software-Dominant Organizations." Paper presented at the 31st Annual International Symposium of INCOSE, Virtual Event, 17–22 July.
- McKinney, D., E. Arnold, and S. Sheard. 2015. "Change Agency for Systems Engineers." Paper presented at the 25th Annual International Symposium of INCOSE, Seattle, US-WA, 13–16 July.
- White, B. E. 2016. "A Complex Adaptive Systems Engineering (CASE) Methodology—The Ten-Year Update." Paper presented at the Annual IEEE Systems Conference, Orland, US-FL, 18–21 April.

ABOUT THE AUTHORS

Dr. Sarah Sheard is an INCOSE Fellow, CSEP, and Founder's Award winner. An INCOSE member since 1992, she chaired INCOSE's SaSIWG from 2017-2021. Her many systems engineering publications now include four INCOSE "Best Papers." When she retired in 2019, Dr. Sheard was a systems and software engineering researcher and consultant at CMU's Software Engineering Institute. Previously she worked at the Systems and Software Consortium, at Loral/IBM Federal Systems, and Hughes Aircraft Company. In 2012, she earned her systems engineering Ph.D. focusing on system development complexity from the Stevens Institute of Technology. Now, she has postponed international travel and is folk dancing by Zoom with her husband.

Mickael Bouyaud is a Worldline business architect, a global leader in seamless payments, and technical director of AFIS, the French chapter of INCOSE. He has expertise in payment systems, specializing in deploying acceptance solutions in retail organizations, mobile security, a PIN on mobile solutions, and Android-based Point of Sale. He previously worked in the mobile industry as a 3GPP standard and algorithm engineer for Mitsubishi, then as a system architect for NXP and Ericsson.

Macaulay Osaisai, a system engineer developing sensors-related systems at L3Harris Technologies, has many years of systems engineering experience in the commercial sector. Previously, he developed autonomous sensor systems for seismic and geophysical applications. His expertise is in systems architecture, systems modeling, embedded hardware and software systems, and low SWaP (Size-Weight-and-Power) systems. As an INCOSE member and a senior IEEE member, he is an MBSE evangelist, developing processes, procedures, and training materials for MBSE. Macaulay conducts systems and MBSE training classes, consulting, coaching, and mentoring. Besides systems engineering, Macaulay enjoys music, tennis, and scuba diving.

Jeannine Siviy is a business and technology strategist who recognizes undiscovered possibilities and spearheads practical innovation paths—cutting through complexity and ambiguity and delivering value at speed and scale. She leads SDLC Partners' Healthcare Solutions, where she and her team address persistent systems interoperation, automation, and ecosystem challenges, with one solution earning a Gartner Hype Cycle mention. She previously held leadership and technical roles at UPMC, Carnegie Mellon's Software Engineering Institute, and Eastman Kodak Company. She received engineering degrees from Purdue and RIT and a Caltech certificate in Technology and Innovation Management. A Pittsburgh native, she enjoys its cultural diversity and has a long-standing passion for nature and animals.

Dr. Kenneth E. Nidiffer has over 56 years of government, industry, and academic experience in software and systems engineering. Ken successfully held positions as a Fidelity Investments senior vice-president, Software and Systems Consortium vice president, Northrop Grumman Corporation technical operations/engineering director, Carnegie Mellon's Software Engineering Institute principal engineer, and a United States Air Force colonel. He is currently an adjunct professor at George Mason University. Ken received a chemical engineering B.S. degree from Purdue University, Indiana; an astronautical engineering M.S. degree from the Air Force Institute of Technology, Ohio; his MBA degree from Auburn University, Alabama; and a systems engineering D.Sc. from George Washington University, Washington, DC.

Systems Engineering Roles in Software Organizations Delivering Service Products

Mickael Bouyaud, mickael.bouyand@ingenico.com; and **Brian E. White,** bewhite71@gmail.com

■ **ABSTRACT**
The software industry has been experiencing several transformations. Development teams now often autonomously deliver business capabilities to software service systems. *The Unicorn Project*, a best-seller, tells how a retail company transformed into an Agile and DevOps organization. This paper uses a model from the systems engineering toolset to understand those organizational changes and proposes an evolution of the systems engineering discipline to increase the value provided by this type of organization.

■ **KEYWORDS:** adaptation, agile, DevOps, engineering, management, organization, product, service, software

INTRODUCTION

Software companies comprise units aggregating sub-services into a final product. These units' teams are systems, and the company organization are system of systems (SoS).

Some advocates promote applying systems thinking in management science (Jackson 2019, Senge 2015). We can observe organizations through perspectives built by systems engineering and model a team with people and resources as a system with functions, roles, components, operational contexts, interfaces, behavior, and environmental interactions.

This system model might apply several different ways. For example, it facilitates analyzing interaction change impacts, streamlining communication paths, and improving global performance. One can also explore final system configuration alternatives. A systems approach provides a deeper understanding of the different leadership and management type advantages and drawbacks.

The International Council on Systems Engineering (INCOSE) Systems and Software Interface Working Group (SaSIWG) studied systems engineering roles in a software organization. As a discussion source, the group studied *The Unicorn Project* (Kim 2019), a best-seller promoting DevOps in software companies. The French Association of System Engineering (AFIS) developed a body of knowledge about service systems engineering (SSEBoK) (AFIS 2020). This SSEBoK uses a building block concept to model a team responsible for providing and operating the product-service system. This model enables team interaction and capability analysis and allows one to understand organizational qualities and evaluate them against another organization.

This paper applies this model to company organizations during different book phases (Kim 2019).

A SYSTEM MODEL FOR A SERVICE SYSTEM

"Service-System is a socio-technical system that comprises: a System of Interest (typically, a product expected to provide some type of service by satisfying specified performance, behavioral and operational needs)." (AFIS 2020)

A unitary service entity, called a Service Building Block (SBB) (AFIS 2020) and depicted in Figure 1 on the next page, describes a product-service system. This SBB interacts and exchanges information or services with other service building blocks, products, or consumers. Each SBB brings its capacities to the wider system, the organization. A search for operation optimization can improve global performance (operating cost, selling price, and service continuity).

This model includes people, means, and implicit or explicit governance in the service producer system. It has reaction, autonomy, and self-adaptation capacities. This adaptability permits configuration, function, performance, and need modifications in long- or short-term objectives.

A team in an organization is a service system building block. We can classify teams and companies as complex systems (White 2015). They have characteristics of ambiguity, unpredictability, evolution, emergence, instability, collaboration, diversity, adaptability, and more.

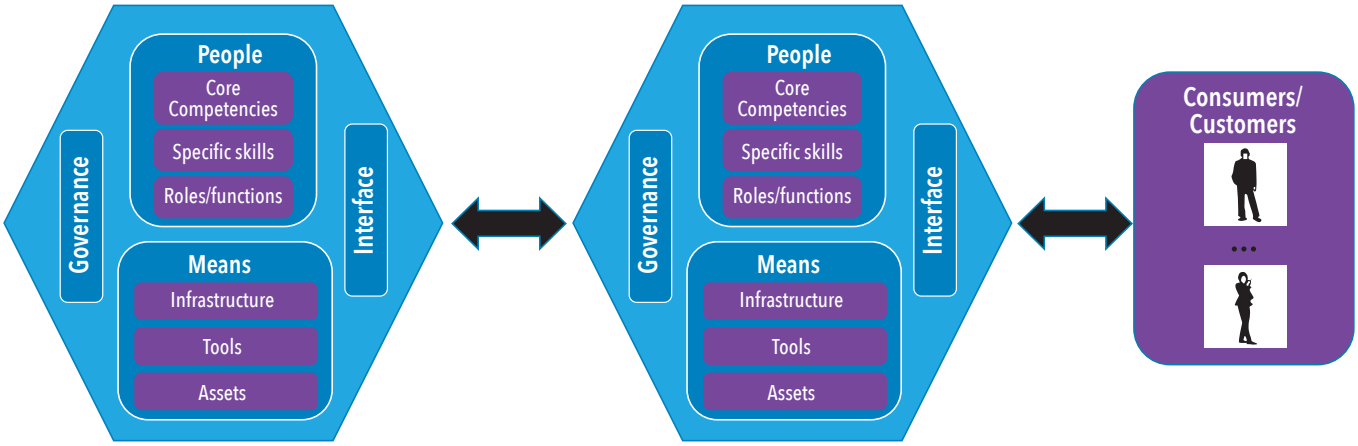


Figure 1. Common product-service building block elements

Various heterogenous software technologies carry services produced by software-intensive organizations. Multiple product-service system components require respective skills, tools, and infrastructures.

The software service industry usually requires fast actions to react to business demands and continual improvement activities extending product services. When usage makes up the demand signal, it can quickly evolve, shortening software development cycles and improving the delivery process. Our attention should be on the integration and delivery process quality and fluidity to make them continuous. Standard tools deploy to merge, test, and deliver freshly developed updates into the other software artefacts.

The Unicorn Project (Kim 2015) describes a retail company's digital transformation through the super-software developer's eyes. This hero leads the organizational adaptation from a pyramidal organization to a horizontal one. The book demonstrates, in a digital world, new organization types better provide service systems. The following paragraphs use the service building block model to give another perspective, showing how a systems engineering perspective can ensure team and organization construct resilience for a software engineering scenario.

THE PHOENIX PROJECT BEGINS: THE MONOLITH ORCHESTRA IS PLAYING

Kim's story begins in a basic retail company organized with a many-tiered pyramidal hierarchy. This company survives on its store permeation throughout the US. There is no need for evolution, no need for a new, challenging project; they only must keep their dominant position.

In this context, basic software programs manage employee services, salaries, documentation, and stock management. Projects are simple and follow classical V-cycle processes. Software teams organize by activities, development, integration, test, delivery, and production with each aspect linked in the chain.

This organization type rations resources and tries to optimize each process by gathering and harmonizing competencies and tools in the same place, akin to Figure 2.

Management attempts firm control and unidirectional interactions and typically operates beyond the other teams. Central management decides tactical orders and relays them through layers much like an orchestra.

Only management and the product itself contact the environment. Each team plays a partition and follows the conductor's lead. Project execution follows a linear and rigid process and imposes a strong coupling between activities for bringing a new capability into the system. This chain will break if any link fails. The lacking autonomy requires the team leader to apply corrective actions directly.

FULL DEVOPS ORGANIZATION OR THE AUTONOMY MYTH

To follow the retail business changes, the company must transform its organization from physical store to virtual store using the Internet to sell goods. Despite its

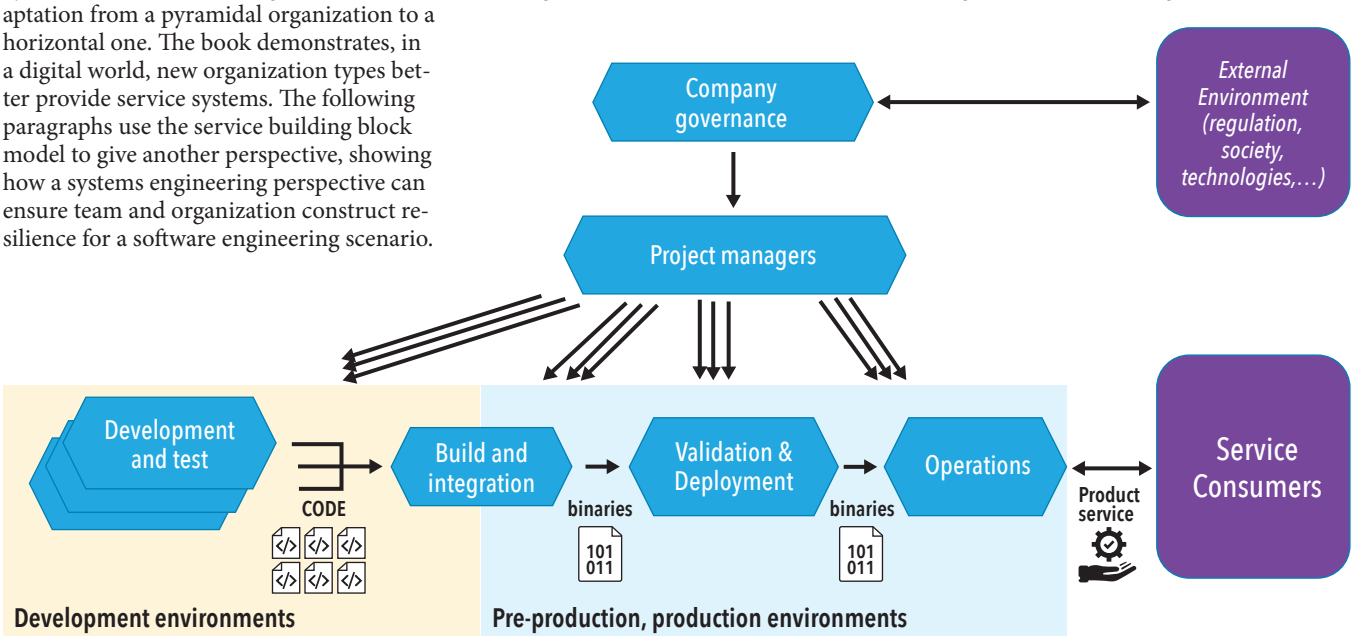


Figure 2. The orchestrated organization

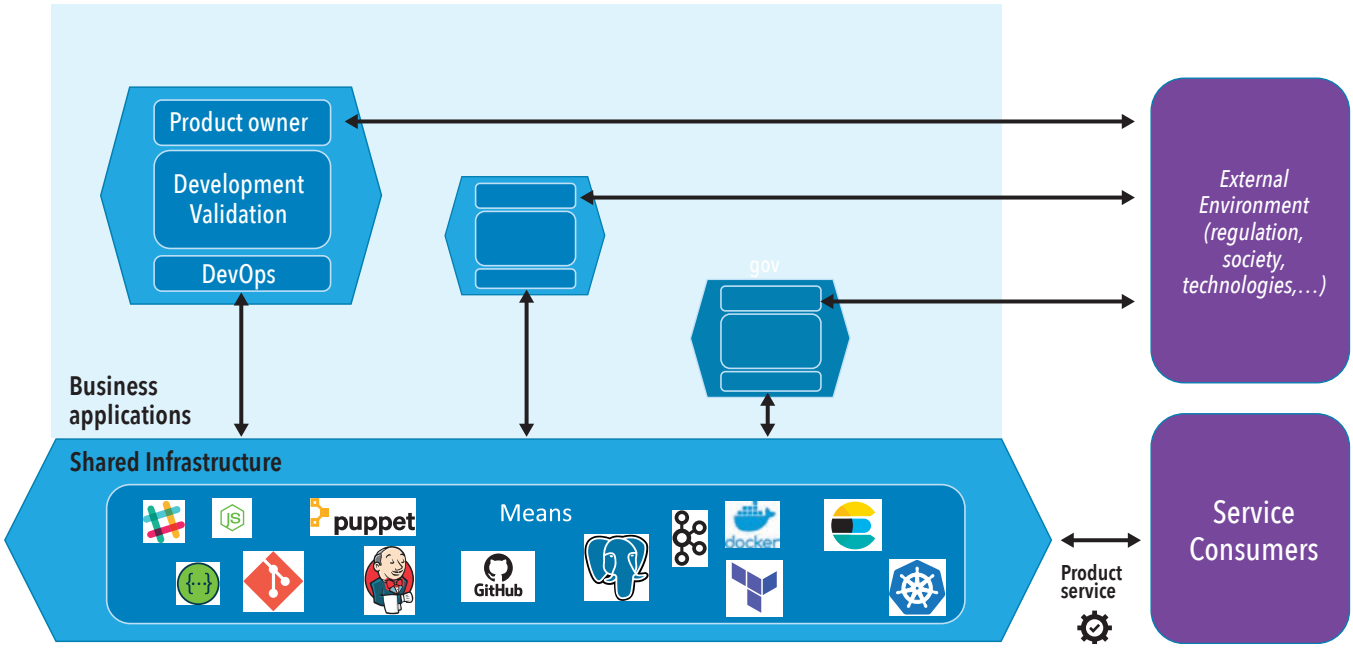


Figure 3. DevOps type teams are autonomous as musicians in a jazz band play music

apparent simplicity, eCommerce systems, such as commercial transactions conducted electronically on the internet, require multiple software technologies, frameworks, and infrastructure typologies to deliver state-of-the-art business capabilities (Akbari 2016). The drawbacks induced by the hierarchical organization for developing and delivering such a system caused the company's implosion.

In summary, a natural-born software hero named Maxine saved the sinking ship by bringing DevOps to the organization. Then from hero to guru, she demonstrated how the subsystems composing the service systems are fundamentally different. The company had to reorganize itself into independent teams providing uncoupled unitary service and creating the final product service system, as several service building blocks create a product service (AFIS 2020). Each service building block can adapt to the environment and, therefore, to the business needs.

At the end of its transformation, *The Unicorn Project* organization applies agility and encourages teams to deliver capability autonomously. In short, development teams become more dedicated to the business. They engage as stakeholders in the company.

With a product owner and DevOps, teams have new environmental interfaces to interact with customers and directly use the appropriate service operation infrastructure. The producing team performs the qualification process during an actual case demo—an agile term for the ceremonial process of testing a delivered capability as an end-user could do. This process efficiently discovers lacking capabilities and bugs and encourages the higher performance of the developers, who behave more as engineers than as technicians like in the previous story period. Outside managers drive teams less, and inside leaders, who help develop quality reference code, push more.

Software infrastructure automates all the processes from building, code transformation, and libraries into program artifacts to run—a term for executing applications in a production context. The Figure 3 infrastructure domain illustrates tools listed in *The Unicorn Project* by their iconic logos. These tools compose the infrastructure and technology stack. The framework and tool landscape is vast and evolves fast. Since Kim wrote the book, some tools are already obsolete. Software engineers use their various communities to follow the trends. Maxine, *The Unicorn Project* hero, spends time on social networks to exchange with experts.

The DevOps individual masters both the delivery flow and the company's infrastructure stack, making the business applications run without any supplementary actions. Systems engineers must know what capability to offer the producing team; how it can improve development, integration, testing, and running processes; and the needed supplemental operations, resources, and operation monitoring.

This organization has many similarities to jazz music, played by autonomous musicians, each having an appropriate feeling and bringing their part to the music. A leader gives the band directions but allows space to empower colleagues for a better performance. This is a similar ideal for an agile, effective organization.

Although we deem this model beautiful, not every company stakeholder may fully appreciate it. It promotes short-term consumer satisfaction, taking shortcuts, possibly interfering with other domain concerns. For example, in *The Unicorn Project*, the author mentions the security teams' regular opposition. Opening an internet protocol (IP) port can assist in a brand-new application, but it provides an open door for any hacker. Where is the forum for this discussion? As in business, the team could manage the security needs internally. A commonly deployed trend not explored in the book is extending DevOps to DevSecOps. What about the General Data Protection Regulation (GDPR) and other regulations? Should DevSecOps be a DevSecRegOps? And what is next? DevOps might explode due to the responsibilities. To avoid a new bottleneck, security and risk leadership and management must remain a transversal activity. Using the SBB model identifies the DevOps roles as a team interface. It enables an organization to more easily and effectively navigate evolutions and identify service block needs in governance and people.

SBB compositions analyze relations between organization components. Applying the SBB helps manage growth, identify potential mean factorization, and find the correct balance between service blocks to deliver a better product service. This system model is a decision tool when some consistency problems arise concerning the company strategy. Increased size might require some additional disciplines to control the whole system and its architecture.

Systems engineers put value in service system design by considering the building system and teams as a system of interest part, analyzing interfaces with the enterprise environment, and studying coupling between teams and components, creating the final product. ■

APPROACHING THE IDEAL, THE AGE OF COLLABORATION

The SaSIWG selected *The Unicorn Project* because the subject company's promoted organization seemed quite distant from systems engineering standards. The SaSIWG intended to understand software engineering culture, the motivation, the reason for the agility hype, and the systems engineers' potential place in a software-driven industry. The conclusion is optimistic. There is room and the need for a systems engineer who will give consistency and resilience to the whole system. The exercise concluded with a live discussion with the author, who agreed DevOps are experts, but today's complex systems require a discipline able to be transversal to the different software engineering domains.

All this reminds us the produced system tightly binds to its producing organization. Finally, service building blocks are product-service systems.

INCOSE increasingly discusses engineering service and social systems. Systems engineering has all the tools to demonstrate its added value in the software industry. We need time and opportunities to make greater impacts. This consideration becomes even more evident when both produced systems and teams embed more artificial intelligent components able to evolve based on their environmental perception. ■

REFERENCES

- AFIS. 2020. "Service Body of Knowledge." http://www.ssebok.afis.community/.
- Akbari, S. 2016. "The Analysis of the Complexities of E-Commerce Industry." Paper presented at the 10th International Conference on e-Commerce in Developing Countries, Isfahan, IR, 15–16 April.
- Jackson, M. C. 2019. *Critical Systems Thinking and Complexity Management*. Hoboken, US-NJ: Wiley-Blackwell.
- Kim, G. 2019. *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. Portland, US-OR: IT Revolution.
- Kumar, A., K. V. Nori, S. Natarajan, and D. S. Lokku. 2014. "Chapter Ten – Value Matrix: From Value to Quality and Architecture." In *Economics-Driven Software Architecture*, edited by I. Mistrik, R. Bahsoon, R. Kazman, and Y. Zhang, 205-240. Burlington, US-MA: Morgan Kaufmann.
- Senge, P. 2015. *The Fith Discipline: The Art and Practice of the Learning Organization*. New York, US-NY: Penguin Random House.
- White, B. E. 2015. "On Leadership in the Complex Adaptive Systems Engineering of Enterprise Transformation." *Journal of Enterprise Transformation* 5 (3): 192-217. Supplementary Material (Appendices): http://www.tandfonline.com/doi/suppl/10.1080/19488289.2015.1056450.
- ———. 2020. *Toward Solving Complex Human Problems*. Boca Raton, US-FL: CRC Press.

ABOUT THE AUTHORS

Mickael Bouyaud is a Worldline business architect, a global leader in seamless payments, and technical director of AFIS, the French chapter of INCOSE. He has payment systems expertise, specializing in deploying acceptance solutions in retail organizations, mobile security, a PIN on the Mobile solution, and Android-based Point of Sale. He previously worked in the mobile industry as a 3GPP standard and algorithm engineer for Mitsubishi, then as a system architect for NXP and Ericsson.

Dr. Brian E. White received his Ph.D. and M.S. degrees in Computer Sciences from the University of Wisconsin and his S.M. and S.B. degrees in Electrical Engineering from M.I.T. He served in the US Air Force, and for eight years, was at M.I.T. Lincoln Laboratory. For five years, Dr. White was a principal engineering manager at Signatron, Inc. In his 28 years at The MITRE Corporation, he held various senior professional staff and project/resource management positions. He was MITRE's systems engineering process office director, 2003-2009. Dr. White retired from MITRE in July 2010, and has since offered a consulting service, CAU← SES ("Complexity Are Us" ← Systems Engineering Strategies). He taught as an adjunct professor at several US universities and currently tutors students in basic mathematics, calculus, electrical engineering, and complex systems. He edited and authored several published books and book chapters, mostly in his book series on complex and enterprise systems engineering with Taylor and Francis and the CRC Press. He presented a dozen tutorials in complex systems and published over one hundred conference papers and journal articles in complex systems, systems engineering, and digital communications over his 55+ year career.

A Complex Adaptive Systems Engineering Methodology

Brian E. White, bewhite71@gmail.com; and **Mickael Bouyaud,** mickael.bouyand@ingenico.com

■ **ABSTRACT**
In recognizing a system's complexity reflects human complexity, this treatise suggests ways to achieve effective progress in complex systems engineering by intentionally including people as an integral system part to develop or improve. We apply essential and relevant multi-disciplinary techniques in addition to the necessary enabling technologies. However, we focus on human aspects making or breaking the system. Although this methodology can apply to almost any endeavor, software engineering is our specific example domain area.

■ **KEYWORDS:** adaptation, complexity, engineering, methodology, modeling, rewarding, simulation, software, stakeholder, system

"Cherish those who seek the truth but beware of those who find it." —Voltaire (Creamer 2021)

"Believe those who are seeking the truth. Doubt those who find it." — André Gide (2021)

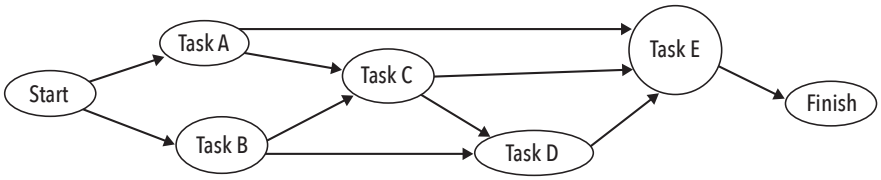
INTRODUCTION

We begin with a background discussion of a simple graphic device (depicted by Figure 1) commonly and historically used to layout system development tasks to determine project schedule "critical paths." We then compare and contrast this device with a modern graphic describing complex systems tasks, activities, and processes.

In Figure 1, we note the following characteristics:

(1) The oval-shaped "bubbles" represent the Start and Finish points and project Tasks (denoted alphabetically, A, B, and on, in order, left to right).
(2) Arrows show links between tasks, at least one connection from Start to a task, and from each task to the next task or Finish.
(3) Every task has at least one input link and at least one output link, while the Start point has no input links and at least one output link. The Finish point has at least one input link and no output link.
(4) The links have no labels.
(5) The graphic is directional (from left-to-right only), feedforward only with no feedback links (return or right-to-left).
(6) We must complete each primary task before considering any other "downstream" task, to its right, complete.
(7) A critical path includes the maximum possible sequence of links and tasks moving left-to-right and Start to Finish. Figure 1 shows two such paths. Therefore, the paths Start-A-E-Finish, Start-A-C-E-Finish, and Start-B-D-E-Finish are not critical paths.

What follows is a detailed and updated Complex Adaptive Systems Engineering (CASE) activity methodology elaboration (White 2021) depicted in the systemigram-like depiction (Figure 2) below (Boardman and Sauser 2008). It reflects many possible or potential interactions among the elements shown. Remember,



Possible Critical Paths:

Start → A → C → D → E → Finish

Start → B → C → D → E → Finish

Figure 1. Historical bubble chart

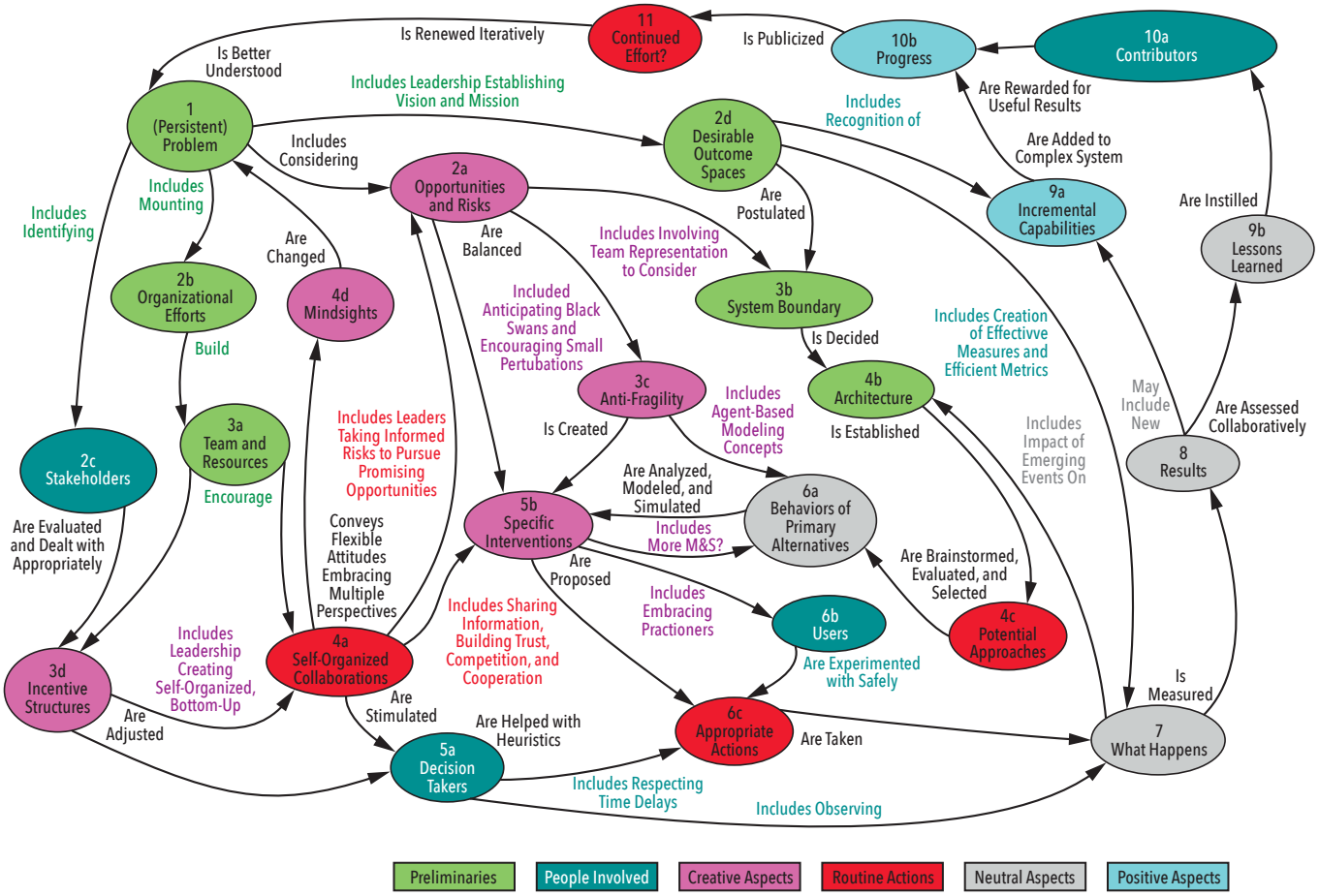


Figure 2. CASE methodology systemigram (suggest revisiting this as you read for a better causality understanding)

in most complex systems, the work never truly finishes because the system continues to evolve independently, despite the engineers' best efforts to "contain" the overall engineering process.

Comparing and contrasting Figure 2 with Figure 1, the Figure 2 graphic has the following characteristics facilitating continual creativity in modifying the methodology's content and flow as we learn more about the complex system:

(1) Bubbles with various coded names depict tasks, activities, or processes, but there is no single starting or finishing point, necessarily.
(2) Arrows link tasks but Start and Finish bubbles are not necessary.
(3) Every task has at least one input link and at least one output link.
(4) Link labels (and labels next to tasks) attempt to indicate how each source task affects the next destination task(s) and what might happen between tasks.
(5) Feedback is essential to accommodate and emphasize evolutionary changes, including recycling or repeating many paths or sub-paths.
(6) Tasks remain open to allow for further exploration or evolution.
(7) There are no special critical paths as defined for the Figure 1 bubble charts.

In explaining the Figure 2 format further, the graphic presents each task with a coded index like 1, 2a, 2b, 2c, 2d, 3a, 3b, to suggest a rough, logical order in purposefully executing the tasks. Each task also has different colors to highlight similar function categories. The category definitions are somewhat arbitrary and are as follows. **Green** is preliminaries while **dark teal** refers to people involved, **purple** to creative aspects, **red** to routine actions, **gray** to neutral elements, and **light blue** to positive aspects. These categories are recommendations and are not critical for understanding the methodology.

Again, as noted above, each arrow linking two tasks has a word or phrase partially describing the interactions between the pair of source-to-sink functions or activities. Also, each label next to a task or link between two tasks briefly describes how each source task acts and affects the destination task with an information flow direction as indicated by the link arrow. A black label indicates how a source activity affects the sink activity(ties).

A colored link label matching the associated source task suggests actions included by particular source activities, especially when relating to the sink task's purpose.

Note this particular diagram is an example, and we do not recommend readers take it too literally. As readers become more familiar with this CASE methodology, it is quite possible to construct a different diagram to depict the intended nuances or emphases better. When creating a new diagram and making the graph easier to read, we recommend keeping the graph planar (with no crossing links).

The methodology, modeled by the Figure 2 diagram, highlights the need for astute leadership and concerns for "managing" a complex system. All system stakeholders have their role in this inclusive methodology. Each stakeholder type will have their unique perspective and system view. This model is available for suggested modification to reflect any perspective not yet represented.

CASE is primarily an engineering methodology, and the work leading up to the current CASE version builds on complexity science as represented by the Santa Fe Institute (https://www.santafe.edu/).

Next, we describe this CASE methodology's aspects, organized by category and addressed by the Figure 2 coded task labels. Although we primarily focus on systems engineering and, more specifically, system of systems (SoS) engineering (SoSE), software engineering is the specific example domain area supporting the International Council on Systems Engineering (INCOSE) Systems and Software Interface Working Group (SaSIWG).

Preliminaries

1—Understanding (Persistent) Problem

As emphasized in the Soft Systems Methodology (SSM) (Checkland 1999), our goal is to understand the (presumably persistent) problem better. However, each understanding level can and should trigger intervening actions to improve the situation. Thus, problem understanding is a continual process requiring re-exercising. It receives information from observation and the (eventual) intervention result. At each such stage, we can delve into the problem further or move on. In SoS or SoSE, reaching mutual understanding across the SoS, especially among component system organizations, is not easy. Thus, this collaboration also warrants extra effort in establishing sustainable SoS mechanisms and pathways for this purpose.

Often, those representing one infrastructure domain, such as hardware in the military or software, lead this problem understanding activity, as in the book the SaSIWG recently reviewed (Kim 2019). However, each relevant domain should have equal opportunities to influence the system's mission.

There are instances where a group within an SoS focuses more on software than on the SoS or systems engineering. This happens within a single organization and across different organizations; it is natural and concentrates effort on needed areas. However, it can lead to "stovepipe" mentalities which are detrimental to overall progress. As a possible remedy, establish open and effective communication between and within the separate groups about the principal problem, especially vital mutual interest topics.

2b—Mounting Organizational Efforts

Individual efforts can address simple problems on a more or less ad hoc basis. Complex issues require additional effort, usually an organizational variety, typically involving multiple organizations. Good leaders, faced with serious problems, can envision, formulate, negotiate, and eventually obtain buy-in participation from relevant organizations, including their own, that can potentially contribute to problem solutions. Initial goals include creating a flexible and resilient inter-organizational structure to ensure, to the possible extent, continual effort from each organization involved. Note the traditional hierarchical authority and accountability tree structure, particularly those of significant depth, may tend to be counterproductive to self-organization (see 4a below).

Compared to hardware teams, we observe software teams are more autonomous and less dependent on discipline-related resources with lengthy availability processes. The "tooling" necessary for executing software tasks is usually quicker to gather than hardware enabling tools. Although systems engineering teams may not be as autonomous, since they are typically responsible for overseeing and influencing several engineering domains, there is also less need for them to wait for systems engineering artifacts to perform their tasks.

Software and systems engineering factions within relevant entities, no matter how autonomous, should strive to achieve effective and efficient organizational coordination. A good way of stimulating this interchange is establishing regular meetings involving a few competent and respected individuals representing each faction. These are people with the authority to commit to decisions reached consensually on the spot rather than causing further delays by separately checking with the bosses offline and negotiating disagreements one by one.

2d—Postulating Desirable Outcome Space

The SoS leadership's initial task is to establish an overall SoSE team vision or mission. This vision should be compelling and easily internalized, motivating and enabling daily personal assessments of the team members' common cause contributions. The team must establish desirable outcome spaces and measures early on to be clear whether postulated solutions, developed later, fit into at least one of these spaces. At this stage, it is premature to focus on specific solutions and their possible outcomes. As the SoSE effort proceeds, these outcome spaces need continual adjustment. One does not want the scope so narrow they miss hazards or opportunities (see 2a below) for good solutions to pursue with informed risk. Nor do they want the scope so broad the problem or solution challenge is far too great.

Admittedly, it is tempting for software designers and systems engineers to rush into exploring new feature developments before thoughtfully and thoroughly contemplating the overall project goals. It is also true software and systems engineering teams are sometimes complacent in not exploring new methods and tools. One painful lesson, learned quite well by experienced systems engineers, is getting too far "over your skis" can lead to much higher costs and greater delays from "backtracking" compared to the more patient approach in considering and deciding upon the various options in moving forward.

3a—Building Team(s) and Resources

The inter-organizational structure initially locates, considers, selects, and assembles talented or qualified individuals who compose the team(s). The job is identifying, qualifying, and obtaining other material and financial resources to support the effort. They plan and place critical processes to ensure a methodology for dealing with unanticipated events. Acknowledging or ignoring this nod to the full situational complexity is likely to positively or negatively impact the whole, respectively. In addition, we must recognize the smooth and inevitable alternative staff member transition into or out of the program for various reasons. Sensitivity to external developments and outreach to others is also important.

Software teams often embody a disruptive culture, an "open source" orientation conflicting with a system solution self-containment goal. Thus, we should discuss what to extract or share with external communities early on.

Leaders should ensure each team includes software and systems engineering experts, at least one of each, who are also good communicators. This will facilitate inter and intra team cooperation in addressing inevitable issues as they arise.

3b—Deciding System Boundary

Contrary to most traditional systems engineering approaches, and perhaps many SoS approaches, the boundary of a complex system is usually both "fuzzy" and evolving. Teams must decide boundaries through thorough discussions while understanding the problem and exercising other CASE activities, including those already discussed and 2a and 2c below. Rationale: As with the 2d analogy above, if the team restricts the boundary too much, it is unlikely they will solve the real problem; if too ambitious, then the difficulties in achieving improvement escalate exponentially. One technique for establishing a working SoS boundary would be assembling and discussing among a small number of representatives from each SoS system component level, hopefully a key stakeholder subset (see 2c below), with authority to commit to actions on behalf of their organizations, as advocated in 2b above. Once this group determines an appropriate boundary, they should alert the SoSE team and their affiliates to help guide their collaborative work. Further, the whole team should agree to adjust the boundary from time to time based on future events.

The main goal is to postulate a reasonable system boundary. When the project involves software, as in most instances, a proper boundary must include all critical software engineering concerns. On the other hand, the project must control or at least address any software engineer's tendency to extend these boundaries too far.

4b—Establishing Architecture

The most critical guide to SoSE is a good system architecture, including the overall software architecture. The team should establish this architecture early in the program with significant effort to essentially guarantee a reasonably stable architecture that does not change much compared to the engineered SoS. Of course, the architecture should change, as appropriate, in response to emergent properties or other unexpected events indicating the need to change direction. In some architectural frameworks, there is a great temptation to create architectural "views" describing specific perspectives to "check a box" required by management. This is fraught with danger if the views promulgate before the underlying architecture fully develops.

There is much to say for today's trend toward service-oriented architectures (SOAs) emphasizing the bottom-line and what usefully delivers to customers. With an SOA, each product becomes a largely autonomous component process aggregate without a strong need to couple them with other behavioral type processes. This generates several advantages, such as deploying these products heterogeneously within other programs involving their own hardware or software resources. Each product has its implicit lifecycle, execution resources, and operation. This is the exact opposite (or at least, philosophical complement) to the detrimental entanglement generated by "spaghetti code" based on a pure imperative coding paradigm application. Even if modularized, imperative coding makes system evolution a nightmare while creating a monolith nearly impossible to operate correctly.

Investment in a good architecture should more than repay its project budget debt by devoting and maintaining a commitment to software quality with decisions ensuring reusability, evolvability, and replaceability through better techniques or technologies.

To the extent possible, the project should "layer" the architecture. Layering dramatically increases flexibility in introducing system and software improvements following changes in the environment or implementation technology. What software might better realize in one era, hardware may do better in the next, and vice-versa. Each layer conforms to closely-knit basic functions, grouped by types, such as applications, networking, communication links, or physical implementations. The interfaces between layers are simple and stable. However, the realization within a given layer can (more easily and often) adapt to different conditions. If the interface(s) to that layer remains(remain) unchanged, the system still operates effectively.

People Involved

2c—Evaluating Stakeholders

There are many stakeholders in a typical SoS due to the SoS level, environmental participants, and those involved with the component systems. As with simpler systems, it is advisable to identify, assess, and evaluate all the key stakeholders to determine who will assist, resist, oppose, or just need the effort and progress updates. The supportive stakeholders require continual nurturing, while those against the project require neutralization or at least marginalization. Psychology, sociology, organizational change management, politics, economics, ethics, and morality are relevant trans-disciplines to apply.

To ensure software and systems cooperation, each entity should identify and seek one or two supporting authoritative individuals to oversee the whole operation and help guide processes conducive to success. If found, established, and maintained, such key stakeholders often ensure a balanced focus concentrated on what concerns everyone.

5a—Helping Decision Takers

"Decision takers," a term more frequently used in the UK than in the US, is a better term than "decision-makers." Taker indicates a more proactive attitude in making difficult decisions. In an SoS, decision taking is more complex due to the numerous stakeholders involved across the SoS level, its environment, and among the component systems and subsystems. Typically, decision-makers take decisions too early in complex systems rather than waiting longer to evaluate the situation better. There are significant time delays due to multiple interactions within the complex system and its environment before the last intervention result becomes apparent. Thus, improvement comes from decision takers waiting until a decision time is more evident. Generally, advisors must provide decision takers with good heuristics (practical rules) to improve decision taking. An example heuristic indicating the need for a decision for any key decision taker within the SoS would be when any component system or subsystem seems to deemphasize the SoS in favor of its system more than expected.

In software, such short-focused engineers are often technicians who make decisions with a closed mindset. This results in inappropriately transferring their responsibilities to toolmakers, managers, or supervisors. Instead, software engineers should take decisions with an open mindsight, and their leaders should more vigorously promote and instill their engineering role.

One key software decision taking aspect is when to release developed code. This should not occur until after thoroughly testing the "beta" versions within the parent system, hopefully with some user participation if testing can occur safely; see 6b and 6a below. In consultation with available systems engineers, software engineers should periodically keep their leaders and managers apprised of the situation and offer valuable advice regarding code release.

6b—Experimenting with Users

Because of uncertainties associated with most SoSs, the SoSE team(s) should experiment with promising ideas in multiple venues. Rather than confining these experiments to laboratory environments typical of traditional systems engineering efforts, teams should embrace practitioners and experiment with users in the field. Users know the operational needs. Thus, leveraging their expertise and experience can make significant progress. This is much better than developing something in a "vacuum" or "throwing it over the wall" and having users reject or misuse the supposedly additional capability. However, teams must perform these operational-type experiments safely so no one is in jeopardy.

Good software engineers make it an ongoing practice to create clear explanatory comments to accompany the executable code. These comments help system users and stakeholders, especially systems engineers, understand the code's intended operational aspects regardless of the software language. This has several advantages, such as illuminating why a software program behaves as it does and helping identify and correct inevitable "bugs." Since only about 30% of the software engineering effort is toward developing code today, systems engineering understanding of their reuse framework, components, tools, and integration techniques is also important.

In addition to the overall software architecture (see 4b above), software engineers should also ensure their relevant software models (see 5b and 6a below, for instance) accompany their executable code. This would further mutual understanding and help everyone keep the big picture of what they develop in mind.

10a—Rewarding Contributors for Useful Results

Rewarding contributors for useful results is the most important CASE

activity to improve system acquisition processes. Too often, giving rewards upfront or via award fees causes programs or projects to fail and accomplishes very little. There must be much stronger incentives to ultimately achieve desired outcomes without having to restart or terminate programs. Reserving rewards for achievement is especially challenging in SoSs, where most component system stakeholders would refuse to join the SoS effort without reaping immediate tangible benefits. Existing incentive structures and reward systems cannot change significantly overnight either. However, with enough resolve, governing bodies could improve the system gradually, perhaps over decades, by making sure more funding and other compensations are later in the programs to help achieve desired results. As we will state again in 3c and 10b below, innovative contracts can help accommodate this systemic change of rewarding for results.

If software engineering lacks systems engineering, there is the danger that software programs will not fully satisfy the overall system needs. On the other hand, participating systems engineers, especially software engineers, should receive appropriate awards for their worthy accomplishments (not just specific software developments).

Creative Aspects

2a—Balancing Opportunities and Risks

Traditional systems engineering focuses too much on risk mitigation. It is more about opportunities with complex systems since the system continuously evolves whether one intervenes or not. Of course, when pursuing more productive pathways to good solutions, it is advisable to do this only with an informed risk plan. Leaders should reward those seeking improvements in this way (see 10a above), even if they are unsuccessful at first. One needs to protect against Black Swans (Taleb 2007) but also stimulate anti-fragile development (Taleb 2012) (see 3c below). Some risk mitigation efforts concerned with avoiding adverse outcomes can lead to promising positive results and vice versa. The most important principle to observe is balancing opportunities and risks. Maintaining reasonable balances among various competing factors, instead of separate suboptimizations, is fundamental in SoSE. One way of ensuring SoS adaptability is to put into place, in advance, a shared management process to use when unexpected events occur.

Software development often naturally inclines toward opportunities, especially in new feature development. While pursuing such further capability, however, those involved must remember how to safely and efficiently integrate new code into the overall system, know potential risks and how to mitigate them through applying sound systems engineering techniques.

3c—Creating Anti-Fragility

First, the SoSE team should protect the SoS from rare catastrophic events (Taleb 2007). For example, suppose the primary approach is no longer viable. Then one of the backup approaches the team carried along may become primary. Also, on a smaller scale, it is a good engineering practice to focus on what might not work or what might go wrong, and have a fallback position involving subsystem redundancy. Traditionally, systems engineering generally assumes everything will work as intended, but this is a flawed assumption with complex systems (Perrow 1999). Once such protections are in place, the SoSE team should subject the SoS to small random perturbations to increase its resilience, robustness, and strength (Taleb 2012) and enhance the SoS's ability to achieve improved "balance" against future adversities. For example, ensure acquisition contracts are broad enough to admit a wider vendor selection, increase competition, and offer opportunities to pay for results rather than perceived promises. This "stirring the pot" helps ensure the best results.

There are countless ways for software to fail or not live up to its intended promises. After drafting each code section, the mindful developer should be willing, if not eager, to take an adversarial approach and spend time thinking about and protecting against misuse and possible, though perhaps unpredictable, anomalies. Some software validation strategies, such as chaos testing, include this process.

3d—Adjusting Incentive Structures

People tend to behave in ways strongly correlating with how we measure and reward them (see 10a above). In some SoS environments, particularly those involving military system acquisition, many stakeholders leading SoS programs focus on the short-term. They change jobs every two to three years and are not responsible for their previous assignments after their reassignment. This CASE activity advocates for, and hopefully helps achieve, positive changes in incentive structures to better facilitate (1) leadership styles creating conditions for self-organization (see 4a below), bottom-up efforts, and discouraging autocratic, hierarchical, top-down approaches; (2) informed risk-taking in pursuing promising opportunities (see 2a above); and (3) more integrated career accountability.

It may be a non-sequitur to envision good software or systems engineers as not incentivized. They most likely take significant pleasure from what they do, especially when seeing the positive effects of their work efforts. However, they should also expect to receive material rewards for making their systems successful (see 10a above).

4d—Changing Mindsights

A principal concern associated with the systems engineering practice is having a "mindsight" conducive to significant progress in complex domains. Traditional systems engineering mindsets focus on requirements, reductionism, and optimization. These mindsets will not work well in the more difficult situations usually associated with complex systems. Mindsight conveys a more flexible attitude established and modified by embracing multiple perspectives of the underlying truth forming the complex systems engineering (CSE) base. Through self-organized collaborations, including individual view exchanges, a better understanding of the problem and what to do is likely. Nevertheless, recognize fully capturing the underlying truth will likely remain elusive.

Considering an outstanding software system, and its many ideas including feature creation, development, change, maintenance, evolution, and retirement, as one of the most important systems engineering domains, is an excellent mindsight to embrace. Systems and software engineers have much to learn from each other.

5b—Proposing Specific Interventions

There are many occasions when the SoSE team is almost ready to try something else to help the SoS advance in the desired direction. We should view these actions as interventions with uncertain outcomes because one cannot predict what will happen in a genuinely complex system. Before implementation, the team should propose each intervention to key stakeholders and obtain their reactions, leading to some plan alterations. The stakeholder may also indicate some additional modeling and simulation (M&S); see 6a below. Finally, before fully committing to a path, some experimentation would also be advisable (see 6b above).

Software developers should resist jumping into feature creation too seriously before telling other stakeholders their ideas to solicit further inputs and ideas influencing the eventual application outcome. A systems engineer might be a worthwhile colleague to approach in this regard.

Routine Actions

4a—Stimulating Self-Organized Collaborations

Early in any process for confronting a problem and seeking improvement,

self-organizational efforts are appropriate. A leader or manager takes charge and assembles an initial action team (see 2b above). The team collects or requisitions resources (see 3a above) and establishes an overall vision or goal for problem resolution (see 2d above). During this time, the team must establish a healthy collaboration spirit among the participants to help facilitate (1) information sharing; (2) building trust; (3) developing individual perceptions, viewpoints, and opinions; (4) cooperation within and across teams; and (5) competition among teams. Collaboration is what enables the self-organization progress to deal effectively with the situation and achieve a solution. This is especially difficult in SoSs, as the component systems have their organizations, each naturally resisting routine collaboration with other organizations because self-interest naturally dominates the affinities with SoS objectives.

At this relatively early stage, software and systems engineers should seek each other out, get to know one another, and prepare to establish stronger relationships as the work proceeds. This activity also stimulates leadership opportunities for anyone involved, even temporarily given the various team members' particular talents and inclinations.

4c—Brainstorming Potential Approaches

Brainstorming is too casual for this critical activity, but it conveys the proper meaning. Here the SoSE team, hopefully in a high-performing, collaborative state, shares ideas about potentially solving the problem, mainly from a technical viewpoint. However, they must still consider the non-technical aspects and all the applicable trans-disciplinary areas. As in typical brainstorming, the ideas should flow freely before anyone on the team attacks ideas. This is where creativity should reign, potentially modifying the Figure 2 CASE methodology systemigram. Then the more evaluative phase begins. The team criticizes, rejects, or refines the offered ideas. The remaining potential approaches should fit within the agreed vision/mission, desired outcome space (see 2d above), and system boundary (see 3b above). Finally, there should be decisions on which approaches to pursue vigorously or bring along with lesser degrees as backup options.

This process provides a great opportunity and solid foundation for collaboration where software and systems engineers can better understand each other's language and term usage. Unless they come from a software background, most systems engineers are unlikely to appreciate software terminology or the pros and cons of software development languages. Similarly, software developers may want to move ahead with their preconceived implementation processes without paying much attention to alternative approaches to reach better solutions. One systems engineering role helps make the evolving software process more adaptable. There seems to be a fundamental cultural difference between software "agility" and the so-called classical (and no longer so influential) "waterfall V" development cycle previously associated with systems engineering, which we moved beyond with SoSE and other CSE forms.

6c—Taking Appropriate Actions

Complex systems operating where they should, at the edge of chaos, continue evolving whether one intervenes or not. Therefore, before taking further action, decision takers should objectively observe what occurs over time (see 5a above). Interventions are necessary; they are what decision takers expect to do. The action takers should take these actions in pursuing an opportunity while remaining informed of potential risks (see 2a above). Whatever actions the action takers within an SoS take, it is appropriate to describe these actions to other key stakeholders at the SoS level, within the SoS environment, and across the component systems or subsystems. Then they have increased abilities to consider their actions, hopefully improving the SoS situation. This information sharing is not as typical in traditional systems engineering environments, where it seems organizations punish, rather than reward, information sharing across organizations (see 4a above).

Good leaders will ensure all team members learn, via communication with their team representatives, the more important interventions and their status while waiting for results. Software engineers may respond to these inputs by providing helpful feedback on their assessments on the intervention's potential success. Everyone involved should try to be flexible in considering their future actions.

11—Renewing Continued Effort?

Our work is never entirely finished in a "healthy" complex system because the system continually evolves. Checkland's SSM (Checkland 1999) already established this. One should view CASE as an iterative process revisiting several or all activities at various times, such as during each activity cycle or after cycling through all activities. The SoS level stakeholders should consider renewing the overall SoS at appropriate milestones. It may be necessary to apply renewed effort on some SoS portions involving one or more component systems or subsystems. Such instances can apply CASE again on a smaller scale.

Contemplating continued efforts is another activity greatly benefiting from the software and system leaders' attention, wisdom, and advice on the achievement-effort tradeoffs especially when there is significant pressure for further change in improving the code or system capabilities.

Neutral Aspects

6a—Modeling and Simulating (M&S) Behaviors

After the SoSE team selects a few viable approaches, a primary alternatives analysis phase begins. M&S includes standard theoretical, analytical capability augmenting methods. To the extent a complex system can decompose into adaptable parts, M&S can help characterize the interactions among these system components and their environment and better determine the factors causing such adaptation. In complex systems where people are part of the system, intentionally, one can benefit from a complementary form called agent-based modeling. This involves postulating a small rule set that autonomous, independent agents follow while interacting with other agents within their hypothetical system environment. An SoS provides a fantastic opportunity for this, considering the numerous and various stakeholders at play. Thousands of iterations, including tens or hundreds of agents, can occur with only modest memories and computational power. Chances are, we can learn much from the behavioral results emerging from these exercises. We can modify the agent rules after adding or subtracting rules while seeing which rule sets seem most effective in illuminating what happens, for example, to agent behaviors, at least to the extent possible. As in more traditional M&S activities, the outcomes inform the SoS development or improvement.

Modeling and simulating people through software programs should be an intriguing idea the more creative software developers might pursue. So far, the popular and prevalent model-based systems engineering (MBSE) techniques have concentrated mostly on technology and the internet of things (IoT) without making much progress on handling SoS stakeholders.

7—Measuring What Happens

Fundamentally, the SoSE team must want results that fit into the desired SoS outcome space. There should already be measures determining whether outcomes fall within that desired space. Better yet, each measure should include readily available metrics for gathering relevant data. For example,

component system or subsystem managers might report their contributions to the SoS level along with why they think each contribution will fit into the SoS's outcome space, and the SoS level would record and share those instances. Teams should ensure whatever data they gathered help avoid wasting resources.

Software and systems engineering leaders can weigh in, look at the data, and learn what valuable information, and hopefully knowledge, they can obtain. They can then apply these insights to improve the existing management processes.

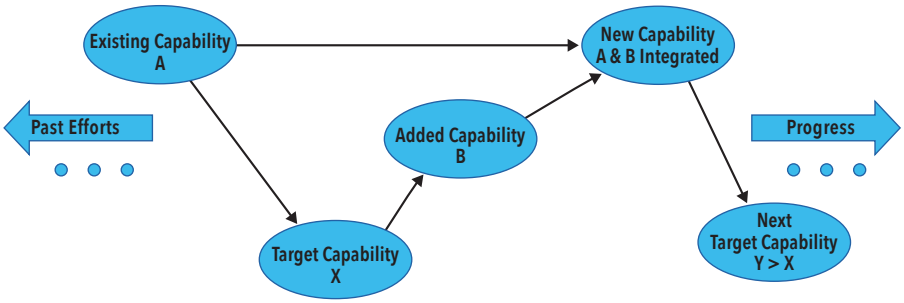


Figure 3. Conceptual process of adding targeted capabilities

8—Assessing Results

Here the main challenge is in reaching consensus across the SoS as to whether they have made improvements. The key stakeholders within any component system or subsystem may disagree on the relative tradeoffs between their local objectives and the SoS level stakeholder objectives. The team should also consult stakeholders within the SoS environment but not directly engaged in the SoS to see whether they noticed improvements. If the SoS provides a public service, the team can assess progress by (1) conducting limited polls or surveys; and (2) contacting selected government officials and lawmakers. A sense of accomplishment would do well toward continuing the improvement efforts.

At this stage, software engineers should view their work objectively in deciding to what extent their programs contributed to realizing a service. In parallel, systems engineers should build on the software subsystem consistency (and hardware) and understand the software engineer's impact, making their systems engineering opinions clear, hopefully with compliments or perhaps constructive suggestions.

9b—Instituting Lessons Learned

Everyone agrees learning lessons is a good thing. Sometimes, with pressing needs to get on with other work, lessons learned are often marginalized or even omitted, and people do not change their behaviors accordingly. Be wary of only a casual or short lessons-learned effort at the program or project's end, for example, giving them "lip service" without calling attention to them verbally or with documentation. We must learn lessons, not just observe them! The essential trick is to retain and institute these lessons on follow-on programs and new projects where those lessons apply. Instilling this activity's importance throughout the SoS is highly advisable, as is searching for lessons from previous projects at the start of each project. More to the point, a systemic process for collecting SoS-related lessons from all component systems would be a good idea. SoSs can share these with other SoSs as well for the benefit of all.

Systems engineers can publicize what they learned from the software engineers and vice-versa. This should strengthen their relationships and greatly benefit their future interactions.

Positive Aspects

9a—Adding Incremental Capabilities

Traditional systems engineering focuses on requirements near the beginning of every program or project and continues after that. This focus often continues because complex system requirements are incomplete, unclear, unstable, or even unknown. In SoSs, it is more realistic to rely on the (presumably already) established mission and desired outcome space (see 2d above) than firm requirements. With each intervention and its aftermath, decision takers assess the extent to which the SoS enjoys additional capabilities—or not (see 5a, 7, and 8 above). If the SoS moves in the positive direction, the subsequent intervention will target an additional capability, as suggested by Figure 3. If not, the decision takers should try something else, perhaps in conjunction with undoing the previous intervention. This process works best incrementally where one builds a little, tests a little, and fields a little. Gradually, with luck, the SoS situation improves.

With these gains, software and systems engineers can take pride in their work if they work faithfully together to benefit all.

10b—Publicizing Progress

Coinciding with result rewarding is publishing not only the recipients but also the results themselves. These publications do not need details; they could mimic stock market quotes, crude oil and gas prices, or public media reports. This serves another purpose—increasing others' motivation to invest in accomplishing similar outcomes (see 3d above). Investments by the component system or subsystem stakeholders in an SoS are necessary in a systemic process rewarding results only. We recommend (1) contractors not receive up-front funding with contract awards; (2) contractors' promises to deliver the "goods" are recorded and held by SoS managers until those contractors actually deliver results to the field; and (3) only successful contractors receive reimbursement and bonuses (see 3c above). Those successful contractors would develop "deep pockets" and be able to continue investing in new projects. The contractors unable to deliver would "dry up and go away." Individual contributors within organizations tend to receive raises, bonuses, and promotions only after the fact, so this natural philosophy is merely extending upwards (see 10a above).

Software and systems engineers should celebrate their collaborative successes by publicly recognizing their joint achievements and highlighting significant increases in system capabilities while thanking those primarily responsible for those gains in performance.

CLOSING REMARKS

As any Complex Adaptive Systems Engineering (CASE) methodology reader has undoubtedly realized, the various individual functional activities highlight somewhat distinct aspects but are interrelated. They overlap and correlate among themselves. There are many ways to proceed, but to apply CASE effectively, one needs to intentionally view all key system stakeholders as being inside (not external to) the system. We must understand human behaviors and characteristics and try to move toward realizing idealistic principles in our efforts.

This write-up tries to help software and systems engineers work well together. As a final comment, we stress complex systems engineering requires exemplar leadership much more than good management, although these practices are complementary (White 2015). ■

REFERENCES

ABOUT THE AUTHORS

> continued on page 46

System Test Approach for Complex Software Systems

Chandru Mirchandani, chandru.mirchandani@gmail.com

■ **ABSTRACT**
There is a need to develop a system approach to develop an adaptive model (Mirchandani 2010) by which the changes in the testing and upgrade or 'problem-fix' processes for complex software are monitored in real-time and incorporated in the development of reliability models for software systems. As system engineers it is our goal to develop, elegant designs that will implement the required capability, i.e., produce the intended result (IW2021 – BT), be both robust and efficient, and limit unintended consequences. Towards this goal, system engineers strive to minimize technical debt and maximize the relevance of the needed capabilities for successful implementation of the system. If one was to follow the classical system engineering process, the system must meet certain needs. If there was a need for the system, be it perceived or an actual problem, the system engineering process, which is constantly evolving, must be enhanced to meet the needs of the 21st century.

It is obvious that testing is necessary to validate that all the critical processes are functional and satisfy the system requirements allocated to software. However, there is a risk if the software is tested to a perceived operational environment based on theoretical workloads.

■ **KEYWORDS:** Software Quality; Defect Find Rate; Failure Intensity; Test Profile

INTRODUCTION

Long, long time ago in a technology landscape not too far away there existed silos of engineering expertise. Unlike good object-oriented design, there was little or no cohesion between these silos even though their individual success was imperative to develop a cohesive interrelated whole for the success of the mission. Why was this? Very succinctly a lack of trust and communication. It is necessary to improve the communication between these silos to accept and share the overall system level objective and how it meshes with the different engineering disciplines, i.e., share and compare the views for the common objective.

This article proposes incorporating the 'actual operational' environment when developing metrics. Since the closer the test environment is to actual scenarios, the more effective the test is in demonstrating the dependability of a system. (Hecht et al 1997) states that software failures are due code deficiencies, i.e. faults, which when triggered by data or a computer state that causes execution of these faults result in failures. The test environment must generate a large number of triggers if it is to be effective for demonstrating the dependability of a high assurance system). This can be actual values or predictions based on actual operational scenarios and data.

The operational scenario must be modelled more accurately by taking into account non-technical data points such as available resources, budget and schedule to weight the data to predict a test strategy. This would act much like the familiar Bayesian statistics approaches (Okamura et al 2006) capable of estimating software reliability in cases where detected software faults are removed. The proposed model will also provide a capability of updating the reliability predictions based on new information being made available such that at the end of the test process, (Hu et al 2008), not only will we have results with a higher level of confidence in meeting the requirements but also a more mature reliability model for future programs.

BACKGROUND

The Quality of Software, (Bach 2003), states:

"Software quality is a simple concept, at least in the textbooks. Just determine your requirements, and systematically assure that your requirements are achieved. Assure that the project is fully staffed and has adequate time to do its work. Assure that the quality assurance process is present in every phase of the development process, from requirements definition to final testing". However, as Bach puts it, this is not so easy in the field. Requirements change, staffing profiles change and most importantly, we have to remember that there's lots of money to be made if you can sell the right product at the right time, or even something close enough to being right. Behind the veneer of metrics and Ishikawa diagrams, quality is just a convenient rendezvous for a set of ideas regarding goodness. As (Weinberg 1991) says, "quality is value to some person".

The system engineering paradigm for

testing implies that a system requirement should be testable, the test should be repeatable and the test method should be validated and the requirement should be verified. This entails that metrics are carefully selected to ensure that these criteria are met. It is the intention of this paper to develop criteria for evaluating software quality, which defines the reliability and dependability of the software. Many factors influence software reliability including the software development processes, the complexity of the system and software requirements, experience of the software developers, the development environment, and the amount and thoroughness of testing. These factors are incorporated into a software reliability modelling approach based on the analyses of corrective action field data collection and failure recording of our software development and testing experience.

(Cai et al 2003) suggested the use of controlled Markov chains be used to synthesize the required optimal testing strategy and adaptive testing strategy. Experience shows that the fault intensity rate (faults discovered per month) follows a Weibull distribution over calendar-time. Each build follows a separate Weibull distribution with its own time scale. For the added code, its time scale is adjusted such that it is considered as the starting or zeroth month. All of the builds' fault intensity profiles are added together to create the composite curve.

PROPOSED MODEL

The first step in defining the model parameters is to describe the parameter and identify the metrics used to measure them. For example, the metric used to measure the software quality is the inherent reliability of the untested code. This means that after some level of testing the failure density of the code reaches a value that more or less remains constant over the operational life of the system. (Musa et al 1987) defines the following mean fault densities at the beginning of various testing phases as follows:

- Coding Phase: 99.5 Faults per 1000 lines of Source Code
- Unit Test Phase: 19.7 Faults per 1000 lines of Source Code
- System Test Phase: 6.01 Faults per 1000 lines of Source Code
- Operational Phase: 1.48 Faults per 1000 lines of Source Code

Thus having laid the ground rules, it is the intent of this model to flag the deviations in defining and measuring metrics to evaluate and track software quality.

Inherent Fault Density:

Even though the literature (Musa et al 1987) states that inherent fault density is the number of faults per instruction, which translates to the fact that greater the program size, the greater the number of faults. From an Operational standpoint, the number of faults exposed due to usage is proportional to how much of this code is used in the normal operation of the software. Thus, for example, a very large software program, which is assigned a very large fault density, may never have these faults exposed or uncovered if it is not used very extensively. Thus is it imperative to ensure that the usage of the code is leveraged to modify or qualify the inherent value. The second aspect of assigning the inherent fault density is the fact that the deliverable source code is used to quantify the fault density. Studies referred to in (Musa et al 1987) show that there is good correlation between source code and fault density. However, the data does not completely support the numbers for subsystem and system test. This model can be extended to today's design paradigm of reusing existing libraries to realize functions.

However, if one was to evaluate the size of the executable code in bytes or words and used this to define an inherent fault density, the variability of the programmer skill and experience has a lesser effect on predicting the fault density. For example, the executable size of the software module will be directly proportional to the CPU usage in performing a function and hence the workload of the function over a period of time will determine the fault exposure rate of that module. An executable size of 10K units using 30% of the CPU in performing the same function a 1000 times a month can readily be compared with an executable size of 30K units using 2% of the CPU in performing the same function f2, 100 times a month. The functions f1 and f2 are both needed to provide the overall system functionality.

Thus, the initial fault density, defined as:

$$d_0 = \text{Predicted Faults per Unit Executable Size} \quad (1)$$

Initial Failure Intensity:

(Musa et al 1987) define this parameter in terms of the linear execution frequency f of the program (or module), the fault exposure rate K and the inherent faults ω0. Thus, initial failure intensity λ_0, predicted as $\lambda_0 = f\ K\ \omega 0$. However, the model described in this paper defines the initial failure intensity as follows:

$$\lambda_0 = f\ K\ (d_0\ Exec) \quad (2a)$$

The linear frequency of the f defined by (Musa et al 1987) is the ratio of the average instruction execution rate r and the number of object instructions is in the program (or module). However, this paper uses the executable size Exec to track the actual instruction set so the linear execution frequency for the model, defined as follows:

f = Average Instruction Rate/Executable Size = r/Executable Size.

Initial failure intensity, defined as follows:

$$\lambda_0 = f\ K\ (d_0\ Exec) \quad (2b)$$

The fault exposure ratio, parameter K, estimated from the size of the program, the average instruction execution rate and the fault reduction factor. (Musa et al 1987) determines a value of K from the following relationship:

$$K = \text{failure intensity/fault velocity} = \lambda_0/(f\ (d_0\ Exec)) \quad (3)$$

B is the Fault Reduction Factor or ratio of net fault reduction to total failures experienced as time of operations approaches infinity or in other words at a steady state when the ratio remains a constant. Generally, the number of faults corrected is larger than the number of failures because in the correction process more faults may have been generated. The skill of the person fixing the fault and the skill of the person identifying the actual cause of the failure, i.e., identifying the fault is paramount in keeping this ratio close to unity. Based on a predicted number of failures υ_0,

$$B = d_0\ Exec/\upsilon_0 \quad (4)$$

Typically, the values of B range from 0.925 to 0.993 with an average of 0.955. However, this number can be improved with perceptive fault identification and good fault reporting. If the personnel identifying the cause of a failure can discern common cause faults and report the information correctly and completely, the value of B can be more accurately predicted. From this relationship, the number of failures can be predicted before the start of test. The prediction of the expected number of failures is as good as the estimates for B and d_0.

$$\upsilon_0 = d_0\ Exec/B \quad (5)$$

In organizations where the complexity of software programs and the failure data associated with the developed software system is available and accurate, the value of B can be calculated from actual historical records. This method of obtaining data is more accurate than estimating it. However, it is only as good as the record and the similarity of the new program to the historic data. Nevertheless, it is the best place to estimate an apriori number for B. It should be noted that B is a metric that measures the quality of the software and the accuracy of the initial fault prediction. From equation (3) and (4), the fault exposure rate K is defined as follows:

$$K = (\lambda_0 / B\, f\, \upsilon_0) \quad (6)$$

FAILURES PER UNIT TIME:

Having defined the ground-rules, the next step is to devise a process by which the failures per unit time can be measured and tracked. From the equations modified from Musa's execution time model, the fault exposure rate for a newly developed software system is determined as follows. After the software system has been specified, designed and code developed, the elements of the system are tested first at the unit level and then at an execution chain or string level. The number of inherent faults in the software exposed before integration and testing starts depends on the expertise and experience of the software developer. The more experienced the developer, the greater the number of exposed faults. Initially the inherent fault density is predicted based on the expertise of the software developers and the historical information based on similar software systems and similar software functions. Using historical information, the initial inherent faults for the software of size E, and fault density do is given by:

$$F = d_0\, E \quad (7)$$

As testing progresses, it is imperative to estimate how good the assertion that fault exposure is a good measure of failure rate of the software. In other words, does the exposure of a fault cause a single failure or does the correction of this fault create or inject more faults and hence increase the number of failures. In most software integration and test scenarios, there is an increased propensity of fault injection through initial fault exposure and fault fixes. Research has shown that the ratio of the net fault reduction to total failures, B approaches 0.955 as time of operations approaches infinity. However, it must be stressed that in most cases that initially this ratio is very much greater than one. The aim is to keep this as close to unity at steady state. To ensure that this ratio is reached quickly, the test profile should imitate the steady state profile as closely as possible. This implies that the design of the test profile is of paramount interest in achieving the steady state failure rate.

TESTING PROFILE

The test profile for optimizing a test strategy can be modelled. The test strategy can either be directed towards verifying that requirements are met or it can be directed towards verifying if the critical functionality of the system has been met. Most often, test profiles are developed based not only on technical underpinnings but also to adhere to the budgetary constraints, contract challenges and implementation schedules. It is imperative early in the test planning stage to lay down the constraints for achieving the required software quality. (Yuan and Gu 2006) elaborates that to ensure testing and development phases work in concert, it is necessary not only to allocate adequate time for each of the test phases, but also formulate and follow reasonable criteria.

For example, if the customer desires to have the best quality in the shortest amount of time, which area has to be optimized from a functional standpoint. Does the customer want to have the best quality for the critical functional elements and have a reduced quality factor for the less critical functionality? (Poore and Trammell 1999) puts it very succinctly that the question is not whether to test, but when, what and how much to test. The more cost effective response to this question would be 'yes'. However, if the customer has to satisfy several stakeholders who will use the system and not always to utilize the critical functions, the response is variable. For example, consider a system where the critical function to user A is to ingest time critical data, for example pertaining to the path of a comet, and calculating the trajectory and position of the same for the next time period. Certain user B may require that this information be displayed in a digital 3-dimensional image rather than in ASCII text. The display function is not as critical as the actual trajectory and position data. However, the customer may have to appease the users A and B; even though user B's display function may be a complex software subsystem which could take time and resources to test and be implemented at an acceptable quality level. (Meyer 2008) states as one of his principles, that a test strategy's most important property is the number of faults uncovered as a function of test time.

Having said this, it is imperative once a schedule date for system delivery is fixed, to evaluate the quality of the software that will be delivered. In industry, the objective is to develop a product within budget and schedule. In a research environment the objective is to develop and prototype systems for evaluating with an intent to use new technologies. The experience and expertise accumulated by over twenty years of working in industry and a research environment, the author has defined the quality of software as follows. The quality of the software can be graded on the following four factors: (1) Is the fault exposure rate at the end of system acceptance test acceptable? (2) Are the identified inherent faults in areas of the software code that can cause critical system to fail at a rate that is not acceptable? (3) Are there any optimization strategies that could minimize loss of critical functionality at an acceptable cost? In fact, (Dhavachelvan and Uma 2005) suggest a framework for testing based on complexity. (4) What are the main cost drivers from a software quality standpoint?

USE CASE

To illustrate the process by which these four quality factors are identified and then quantified, a software system was analysed and modelled using the basic premise underlined for phase-based test profiles. In this model, the fault exposure rates are also tied to how the resources are used. The first step in the process is to evaluate the functionality of the software system from a functionality point of view and derive a test profile to optimize the quality factors. These quality factors are then compared against the actual test profile and the quality factors of the actual system to demonstrate the relative difference and validate the advantage of using the prescribed approach.

REQUIREMENTS-BASED TEST PROFILE

A software system used to ingest time critical data and provide positional data very akin to a path of a comet, was developed using redundant hardware and software resources. The switchover and recovery time in the event of hardware or software failures were implemented using innovative sensing mechanisms to detect and fail-over to available redundant hardware and software elements. The customer and the developers agreed to a staged system sell-off based on requirements, i.e., a 75% requirements pass-rate meant that the system was 75% complete and testing to a perceived operational workload. The schedule was fixed and the system budget was optimized to sell-off as many require-

ments as possible to meet the system milestones. Table 1 shows a conceptual application of how a test profile based on this premise can be used to test and deliver the software system. The different projections tabulated in Tables 4 through 6 are calculated using the assumed parameters shown in Tables 1 and 2.

Table 1. Requirements based defect finds

Resources	Phase 1	Phase II	Phase III	Phase IV
Time in weeks	28	44	24	28
Computer Resources	4	4	2	2
Personnel	12	12	6	6
Exposure Rate/week	6	3.25	4.25	1.25

Consider the software system made up of five separate software subsystems, which are of varying size in terms of Sources Lines of Code (SLOC). The testing profile is based on the size of the software systems and testing profile is based on the relative size and the number of requirements that are being tested and passed. The total SLOC of the system was about 275,000 and the relative defect exposures of the subsystems were calculated as shown in Table 3, which are plotted in Figure 1.

Table 2. Relative requirements test exposure rate

Subsystem	A	B	C	D	E
SLOC	5000	25000	85000	120000	40000
Defect Rate	0.02	0.09	0.30	0.44	0.15

Table 4 shows the defects find projection in the four test phases lasting 28, 44, 24 and 28 weeks. These defects are translated to defects per month. These defect rates per month for all the software subsystems at every test phase are calculated, normalized and tabulated in Table 3.

Table 3. Projected defect rates – requirements

Subsystem	A	B	C	D	E
Hand-off	1	2	15	15	7
Phase 1 (7 mth.)	1	3	10	15	5
Phase 2 (11 mth.)	0	2	6	8	3
Phase3 (6 mth.)	0	1	2	3	1
Phase 4 (7 mth.)	0	0	1	1	0

FUNCTIONALITY-BASED TEST PROFILE

In a similar manner, the curves showing the functionality-based defect find rate over the test phases are calculated and plotted. In this case, the predicted defect find rate is based on the functionality allotted to the software subsystems. It is presumed that the stakeholders and the system implementers have agreed to buy-off the system based on the defects exposed based on a functionality and criticality test profile. The relative test exposure rate of the subsystems is shown in Table 4 and the defect rates per month for all the software subsystems for every test phase are calculated and are tabulated in Table 5. Figures 1 and 2 plot the values from Tables 3 and 4.

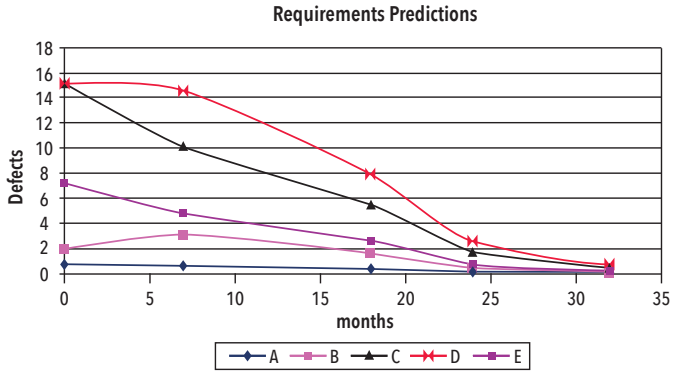


Figure 1. Predicted find – requirements

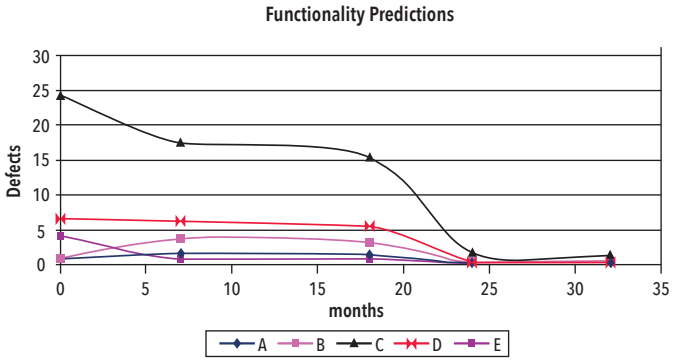


Figure 2. Predicted find – functionality

Table 4. Relative functionality test exposure rate

Subsystem	A	B	C	D	E
Func.	2	3	7	5	2
Ratefunc	0.11	0.16	0.37	0.26	0.11
Crit.	3	2	1	2	5
(Ratefunc) crit	0.04	0.08	0.37	0.13	0.02
Ratenorm	0.06	0.12	0.58	0.21	0.03

Table 5. Projected defect rates – functionality

Subsystem	A	B	C	D	E
Hand-off	1	1	25	7	4
Phase 1 (7 mth.)	2	4	18	6	1
Phase 2 (11 mth.)	1	3	16	6	1
Phase3 (6 mth.)	0	0	2	1	0
Phase 4 (7 mth.)	0	0	1	0	0

ANALYSIS OF DEFECT PROFILES COMPARISON TO RELATIVE DEFECTS FINDS

Finally, the predictions based on requirements and functionality are compared against the defect find ratio to demonstrate the advantages of using the functionality model. Table 6 shows the relative number of defects/month that was found on a system, which had the same characteristics as the analysed conceptual software system. Figure 3 shows curves for the time phased test activities. The actual defects rate for the different software subsystems show a marked difference in the defect rate profiles.

Table 6. Defect find rates – requirement profile

Subsystem	A	B	C	D	E
Hand-off	1	1	10	10	5
Phase 1 (7)	1	2	13	13	6
Phase 2 (11)	0	0	0	1	0
Phase 3 (6)	1	1	11	12	2
Phase 4 (7)	0	0	4	5	1

The requirement-based test profile shows a wide variation in the defect rate through the different test phases as seen in Figure 4. Whereas the projected functional defect rate shows a decreasing defect rate as the testing proceeds, the requirement-based test profile shows a rising defect rate which rises to a maxima and then falls as the test proceeds. Furthermore, in time the test phases the defect rate rises again and then decreases sharply, but not to as low a level as the functional-based test profile.

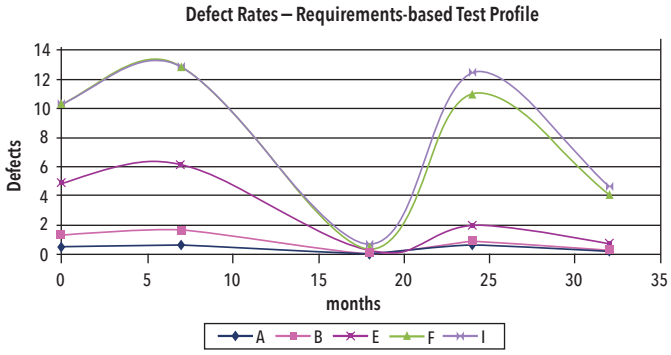


Figure 3. Requirements-based finds rate

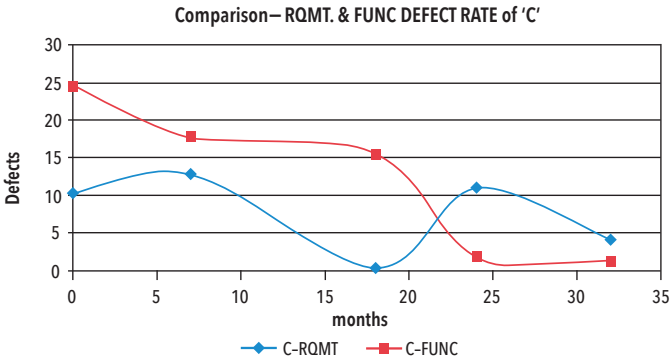


Figure 4. Comparison – rate for 'C'

From the shape of the curve (based on a requirements-based test profile), the defect find rate for subsystem C contrary to the functional-based projections increases sharply after decreasing.

Finally, even though the cumulative defects exposed on a monthly basis is about the same for the functionality-based test profile, the requirements-based test profile rate does not follow the decreasing trend that was projected.

SOFTWARE QUALITY AT SYSTEM ACCEPTANCE

The ISO/IEC 25010 characterizes software quality in terms of functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. This paper supports that characterization by identifying functionality as the most important characteristic. The functional correctness, completeness and appropriateness of the software is paramount in satisfying the stake-holders and achieving system acceptance.

It is an accepted fact that there is a hard schedule requirement for system acceptance; hence, it is important to optimize the testing profile such that the software quality at system acceptance is as good as we can achieve within the budget and schedule constraints. It can be argued that the since the most important objective is to ensure a satisfied customer, a contractor could increase the resources such that the required quality level is reached. However, there is a point of diminishing returns, because there is a limit to the number test platforms and, as in this case, it is sometimes the limiting resource.

Translating the defect rate to a system failure rate, the following curves, shown in Figures 5 and 6, using generic time units and limitless resources, show that defining a test profile that exercises the functionally critical subsystems, ensures a lower failure rate, i.e., higher value on the y-axis, at the time of system acceptance.

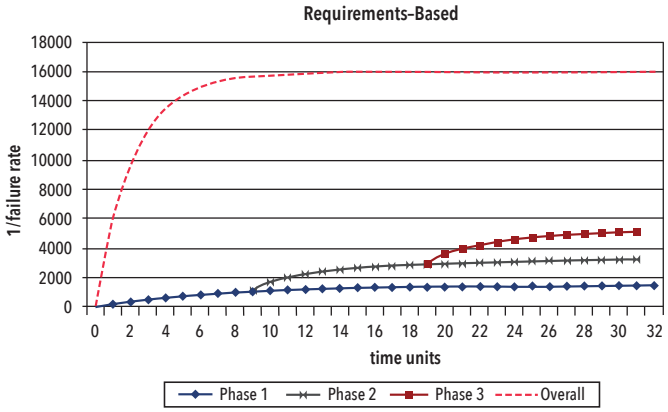


Figure 5. Requirements-failure rate profile

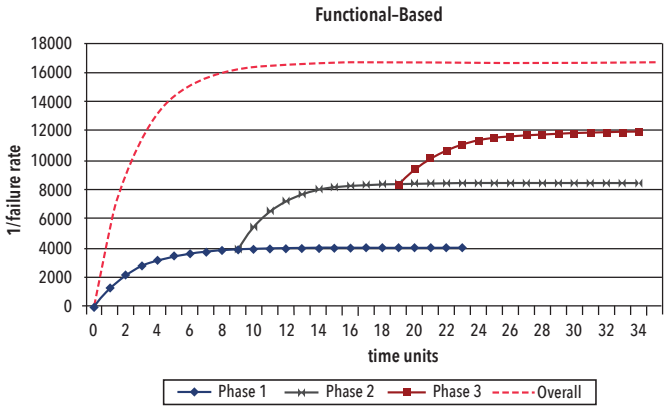


Figure 6. Functionality–failure rate profile

From a system-engineering point of view, we have to ask the question:

(1) *Is the fault exposure rate at the end of system acceptance test acceptable?* This is dependent on what the stakeholder thinks is acceptable. It is the contention of the system design team that if the system meets the objectives for successful operations the system is acceptable. In many cases this means that the stakeholder requires the system to provide the primary services expected of the system when and how needed. The 'when' drives the availability of the system and the 'how' drives the fidelity and performance of the service. The availability of the system is directly proportional to the quality of the designed system, i.e., the hardware architecture and the software subsystems. Given that the hardware quality is

well defined in terms of hardware failure rates and repair times, it is imperative that 'good' data on the failure rate and recovery from failure times of the software is available. Good data is obtained from recorded and reported metrics and in the case of software; it is the defect exposure rate or 'defect find rate' when the system goes operational. Thus, if the projected failure rate of the system has been estimated in terms of hardware and software failure rates, the goodness of the estimation and the allocation to hardware and software thereof is as good as the projections.

It is seen from the analysis, that though the estimation of the inherent defect estimates on the software system, as the whole do not vary appreciably between the different methods of evaluating the same, the defect exposure rate through the test phases does vary. The planned test profile did not correctly project the actual defect find rate, and hence and elevated risk of not meeting the software failure rate requirement.

(2) *Are the identified inherent faults in areas of the software code that can cause critical system to fail at a rate that is not acceptable?* This is best answered by calculating the failure rates and verifying that the failure rate projected at system acceptance meets the critical requirement for the system. The functionality-based defect find rate curves are based on the criticality and functional executable code. From the defect find rate curves it is seen that the actual defect find rate for the identified critical software subsystem does not follow the projected defect find rate, and hence in most probability will not meet the critical requirements of the system.

(3) *Are there any optimization strategies that could minimize loss of critical functionality at an acceptable cost?* The statement of work and the program schedule sets the constraints. Given a hard schedule date, test implementers can only meet a quality level that is defined by the test profile and allowable budget. However, the relationship between the selected test profile and within the allowable budget is not unbounded. For a fixed schedule the resources limit the amount of testing that can be accomplished. This would suggest that there is a limit to the increase in budget for a fixed schedule, and hence a limit to the quality, in terms of defect find rate, attainable.

Budget ≤ Maximum Resources utilized within Schedule
Quality ≤ Maximum Testing performed within Schedule

Any further increase in quality is attainable only with an increase in schedule that would allow more resources to be used. Thus given a schedule there is a limit to the attainable quality; and vice versa, given the required quality level, the minimum schedule needed can be estimated.

(4) *What are the main cost drivers from a software quality standpoint?* From the analysis the main cost drivers to software quality are available test resources and test schedule. The main driver is the test schedule. Given the test schedule, it is important to realize that greater the critical defect finds in the allowable test stages, the more optimal the software system quality. Which leads to the conclusion: If the stakeholder considers system quality in terms of 'how well the system meets the critical functionality of the system', better (or more accurate) the projection for defect 'finds' and higher the attainable software quality within the allowable schedule.

ON-GOING RESEARCH

The author is working on an adaptive model that would better project and dynamically update defect 'find' rate for software based on behavioural patterns. ■

REFERENCES

- Bach, J. 2003. "The Challenge of 'Good Enough' Software." *American Programmer Magazine*. Updated version of the original downloaded from https://www.satisfice.com/.
- Cai, K. Y., Y. C. Li, and K. Liu. 2003. "How to Test Software for Optimal Software Reliability Assessment." *Proceedings of the Third International Conference on Quality Software (QSIC).*
- Dhavachelvan, P., and G. V. Uma. 2005. "Complexity Measures for Software Systems: Towards Multi-agent based Software Testing." *Proceedings of the International Conference on Intelligent Sensing and Information Processing (ICISIP).*
- Hecht, H., M. Hecht, and D. Wallace. 1997. "Toward More Effective Testing for High Assurance Systems." *IEEE Proceedings of the High-Assurance Systems Engineering Workshop.*
- Hu, H., C. H. Jiang, and K. Y. Cai. 2008. "Adaptive Software Testing in the Context of an Improved Controlled Markov Chain Model." *Annual IEEE International Computer Software and Applications Conference.*
- Meyer, B. 2008. "Seven Principles of Software Testing." *Software Technologies, Computer.*
- Mirchandani, C. 2010. "System Engineering Approach for Complex Software Testing." *8th Annual Conference on Systems Engineering Research.*
- Musa, J. D., A. Iannini, and K. Okumoto. 1987. *Software Reliability, Measurement, Prediction, Application*. US: McGraw-Hill.
- Okamura, H., H. Furumura, and T. Dohi. 2006. "On the Effect of Fault Removal in Software Testing - Bayesian Reliability Estimation Approach." *IEEE 17th International Symposium on Software Reliability Engineering (ISSRE).*
- Poore, J. H., and C. J. Trammell. 1999. "Application of Statistical Science to Testing and Evaluating Software Intensive Systems." *Proceedings of the Science and Engineering for Software Development: A Recognition of Harlan D. Mills' Legacy.*
- Weinberg, G. M., 1991. *Quality Software Management, Volume 1: Systems Thinking*, US: Dorset House.
- Yuan, Y., and S. Gu. 2006. "Research And Establishment Of Quality Cost Oriented Software Testing Model." *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)/CCGEI.*

ABOUT THE AUTHOR

Chandru Mirchandani is an INCOSE Fellow, AIAA Associate Fellow, ESEP and IEEE Senior Member. Currently, with Leidos, as a Qualified System Architect, Principal System Engineering Lead in Reliability, Maintainability and Availability Engineer, and SME in the Digital Engineering Center of Excellence. He developed, authored and delivered in depth analysis for system reliability and resilience; and developed an innovative process to distribute & allocate software faults to SW based on CPU usage, content & environment. Concurrently, an Adjunct Professor at George Washington University teaching System Engineering for the past 10 years. Technical chair for national and international symposia and conferences; authored over 40 papers in complex systems and optimization of system concepts; and reviewed technical papers. He has an MEEE (Rensselaer), MS Systems Engineering (University of Maryland), and a PhD Systems Engineering from George Washington University. Interests include research, design and model development of complex systems based on reliability, performance and cost; fault-tolerant systems; Bayesian processes and decision theory. He has over 30 years as a Senior Staff Engineer (Lockheed Martin and heritage companies) in the research, development, design and integration of VLSI-based telemetry systems at NASA Goddard Space Flight Center. Co-Principal Investigator on Intelligent Sensor and Satellite Networks for Earth Science and Exploration. Lead System Engineer and Architect on conceptual designs and concepts for Advanced Traffic Management Systems and Intelligent Vehicle Highway Systems. Awarded a Fulbright Specialist Grant 2012, taught Risk Management in Large Scale Systems at the University of Sri Lanka.

Systems Engineering and DevSecOps: Reviewing the Principles

Dr. Richard Turner, rturner@sei.cmu.edu

■ **ABSTRACT**
As software engineering adopts a more continuous delivery mode for embedded and complex systems, systems engineering must adapt and influence DevSecOps and related practices. In this article, I revisit agile, lean, and DevSecOps principles and comment on their interactions, focusing on those that may increase product and system development, deployment, and evolution risk, and those that increase improvement opportunities through productive engagement across the two disciplines. This material is also available as a Software Engineering Institute Blog Post from 15 March 2021 [https://insights.sei.cmu.edu/blog/comparing-devsecops-and-systems-engineering-principles/.]

ARE THERE FUNDAMENTAL ISSUES BETWEEN THE SYSTEMS AND SOFTWARE DISCIPLINES?

I believe we do not fully understand the interaction of these two disciplines, and experience from early application suggests counterproductive model clashes (Boehm, Port, and Al-Said 2020). The following table identifies fundamental differences between historically practiced systems engineering and systems engineering for evolving software engineering environments. Mitigating the clashes could enhance the DevSecOps adoption success rate and support adjustments to both disciplines. However, mitigation requires identifying the specifics and understanding model clash contexts and sources.

Due to the breadth of domains both disciplines cover, I have revisited theirbasic principles to better understand the model clashes. Systems engineering principles generally focus less on activities than the lean, agile, and DevSecOps principles. Therefore, I present them first and then discuss how the Agile, Lean and DevSecOps principles interact with the systems engineering principles and activities.

SYSTEMS ENGINEERING PRINCIPLES AND ACTIVITIES

The *Systems Engineering Body of Knowledge* (SEBoK) website [https://sebokwiki.org/wiki/SEBoK_Introduction] defines systems engineering as

> *"…a **transdisciplinary** approach and a means to characterize and manage the development of successful systems, where a successful system satisfies the needs of its customers, users, and other stakeholders. Systems engineering focuses on holistically and concurrently understanding stakeholder needs; exploring opportunities; documenting requirements; and synthesizing, verifying, validating, and evolving solutions while considering the complete problem, from system concept exploration through system disposal."*

Systems engineering principles tend to be less specific than Agile, Lean, and DevSecOps software engineering principles because they apply across so many domains. Systems engineering principles also evolve, and INCOSE and other organizations are addressing some of the differences identified in the previous table. However, just as with DevSecOps, the rate of practitioner use of these refined principles is difficult to determine. Here are 14 principles developed by the NASA Systems Engineering Research Consortium [https://www.nasa.gov/consortium/SystemsEngineeringPrinciples](I have highlighted critical concepts for this article).

COMPARING LEAN-AGILE PRINCIPLES TO SYSTEMS ENGINEERING

DevSecOps is an extension of Lean-Agile principles, and depends on their presence for success. The following sections describe each Lean-Agile and DevSecOps principle and provide a brief commentary on its relationship to key, related systems engineering activities. Given the numerous Agile and Lean principle versions, I used the collective principles articulated in the SAFe Scaled Agile Framework for this comparison (Scaled Agile Framework 2020).

Principle 1: Take an Economic View.

We make decisions by comparing clearly stated or unconsciously considered values. In systems development, specifically addressing values allows teams to make decisions in an economic framework (Reinertsen 2020). Value should be a work prioritization and sequencing factor.

Commentary: Understanding and intentionally capturing requirement and design component value as multiple stakeholders see them enables better impact analyses

Systems Engineering as Historically Practiced	Systems Engineering for Evolving Software and DevSecOps Environments
Large-batch processing (products, documents, events)	Small batch processing (products, documents, events)
Single-pass lifecycle (all requirements done before initiating the design; all design done before implementation)	Incremental, iterative multi-pass lifecycle (small product batches and their artifacts built/tested iteratively, delivered incrementally)
Single-point design	Set-based design
Solution intent fixed early (all requirements defined in detail early)	Solution intent is variable early (only near-term requirements in detail; others are higher level with details based on learning)
Fixed point, large-batch integration (components all "done" before integration occurs)	Cadence-based, small-batch integration used as frequently as feasible; integrate as available to prevent rework (for software, this may be daily or continually)
Centralized, command-and-control leadership	Mix of centralized and decentralized leadership; "servant leadership"
Detailed, allocated baseline early; high overhead change management practices remain for the rest of development	Allocated baseline abstraction level allows learning-based change throughout development; no high-overhead change processes
Hardware and software treated separately, integrated late	Hardware and software treated together, integrated early and frequently
Large-batch model-based engineering applied to improve requirement and design detail before implementation; often abandoned after design	Model-based engineering moves between large- and small-batch modeling activities; models and simulations flow with implementation and support the entire lifecycle, development through sustainment
Projective (to be) requirements and design documentation dominates early discussion and activities	Projective documentation takes second place to working prototypes and demos; guides and does not specify; documentation is as-built, not to be.
Systems engineering function separate from hardware and software development functions	Systems engineering function integrated into capability-focused teams including all required disciplines (hardware, software, user experience, and reliability)
Component-based work breakdown structure	Capability-based work breakdown structure
Systems engineering primarily as artifact transformation (Requirements->Architecture->Design)	Systems engineering as a service (facilitating artifact transformation; focus on communication, coordination, conflict resolution, and collaboration)
System architecture decisions neutral to development approach	System architecture decisions strongly support loosely coupled components/subsystems, especially for software capabilities
Assumes early work is correct and late failure is a surprise	Assumes early work is inherently flawed and learning from early failure feeds the evolution of knowledge about the system
Freezes system and software architecture early	Refines an intentionally extendable and iteratively evolving architecture throughout development and sustainment
User participation is only early and late	User participation is continuous throughout the lifecycle

and prioritization in development and sustainment. Using a common value-determination process, including all success-critical stakeholders , can provide decision visibility; support decisions at deeper implementation layers; and support identifying temporal, internal, and external influences impacting value. Appendix C of Boehm and Turner's work (2015) provides a model of value-based systems engineering.

Principle 2: Apply Systems Thinking.

Systems thinking broadens the development focus, encompassing the entire value stream in acquisitional, developmental, and operational organizations (Centers for Disease Control and Prevention 2017). It considers more factors than requirements or how the product system operates, and it enables us to understand the socio-technical system encompassing the product and its context.

Commentary: Systems engineering, by definition, incorporates systems thinking. Understanding the full effort (including the DevSecOps activities and requirements), the associated value streams, and overall value network is critical to system thinking's holistic nature. Systems thinking is an obvious common principle.

NASA Systems Engineering Research Consortium Systems Engineering Principles
Principle 1: Systems engineering **integrates the system and the disciplines** considering the budget and schedule constraints.
Principle 2: Complex systems build complex systems.
Principle 3: A focus of systems engineering during the development phase is a **progressively deeper understanding** of the interactions, sensitivities, and behaviors of the system, stakeholder needs, and its operational environment.
Principle 4: Systems engineering has a critical role through the entire system lifecycle.
Principle 5: Systems engineering is based on a middle-range set of theories.
Principle 6: Systems engineering maps and **manages the discipline interactions** within the organization.
Principle 7: Decision quality **depends on the system knowledge** present in the decision-making process.
Principle 8: Both policy and law must be properly understood to not overly constrain or under constrain the system implementation.
Principle 9: Systems engineering decisions are made under **uncertainty**, accounting for **risk**.
Principle 10: **Verification** is a demonstrated understanding of all the system functions and interactions in the operational environment.
Principle 11: **Validation** is a demonstrated understanding of the system's value to the system stakeholders.
Principle 12: Systems engineering solutions are constrained based on the **decision timeframe** for the system need.
Principle 13: **Stakeholder expectations change** with advancement in technology and understanding of system application.
Principle 14: The real physical system is the only perfect representation of the system.

Principle 3: Assume Variability; Preserve Options.

Locking in a single, detailed description of a system that will take years to develop can become a barrier as soon as a change in one or more naturally evolving factors—threats, political landscapes, economics, technology, or markets—invalidates an assumption or specification. Acquirers and developers must acknowledge variability and uncertainty as facts of life, and investing in and maintaining options and making decisions at the last responsible moment is a good way to manage change (Matts 2017).

Commentary: While specific systems engineering tasks look at risk management, safety, and security-failure modes, fewer activities address understanding how environmental changes impact the actual development, once approved. Identifying and managing useful options to reduce the impact of changes requires ongoing resources and intentional activities.

Principle 4: Build Incrementally with Fast, Integrated Learning Cycles.

This principle provides feedback on estimates, assumptions, and feasibility quickly enough to eliminate high rework costs. Coupled with small batch size, it offers high stability in work planning and enhanced agility to capture opportunities resulting from uncertainty and variability. It eliminates the overhead of maintaining large, monolithic, and generally inaccurate master schedules and focuses on delivering value quickly.

Commentary: This principle is a key concern. Systems engineering generally drives software development and sustainment to the bottom of the traditional V model (Miller 2019). Adaptation to DevSecOps' continuous, incremental, and iterative nature forces an earlier and sustained focus on software-related systems engineering activities reducing the V model risks. The cultural challenge for systems engineering is moving from relatively rare interactions to continuous involvement in software development and evolution.

Principle 5: Base Milestone Completion on the Objective Evaluation of Working Systems.

Traditionally, systems engineering treats milestones as gates, with passage based on static technical artifacts with little completeness or accuracy evidence. Status demonstrations are more useful and provide more learning opportunities.

Commentary: Technical reviews (particularly those supporting milestone gates and progress measurement) often build on boilerplate documentation, overly formalized plans, incomplete or inadequately vetted requirements, or design specifications, including guesses made to remove "to be determined" items rather than acknowledging these items require further analysis at the milestone. The scope is also extensive, driven by complex critical resource scheduling. The minimal viable product (MVP) concept can apply in systems engineering to include demonstrable, measurable outcomes for smaller work efforts. Systems engineers often base their decisions on the analysis, prototyping, and experimentation results; such results could reasonably act as systems engineering MVPs.

Principle 6: Visualize and Limit Work in Progress (WIP), Reduce Batch Sizes, and Manage Queue Lengths.

Visualizing and limiting work in progress regulates the number of tasks worked on simultaneously (Atlassian 2019). It also keeps from overwhelming the human resources by the context switching between tasks. Managing batch size and queue lengths supports the WIP focus with the "stop starting and start finishing" principle since the user receives value only with

completed work, and work waiting in a queue is a waste (Reinertsen 2017).

Commentary: Systems engineering is often understaffed, and the continuous nature of a DevSecOps environment puts a strain on available systems engineering resources. Understanding how much work to expect and its production rate supports maximizing the flow and increasing the value of many systems engineering activities. Staffing practices are a significant factor for systems engineering in applying this principle.

Principle 7: Apply Cadence and Synchronize with Cross-Domain Planning.

Agile-Lean organizations work on a continuous understanding, implementation, and feedback cycle to provide the most value over a cycle with the available resources. Hayes (2017) explains setting cadences and synchronizing across the various teams and activities is the Lean answer to bounding uncertainty and are essential to:

- providing predictable results and feedback opportunity cycle
- aligning metrics
- converting unpredictable events into predictable ones
- providing opportunities to understand, resolve, and integrate multiple teams' work and manage various stakeholder perspectives simultaneously.

Commentary: Predictive or "push" scheduling usually downplays uncertainty and provides reasonable estimates based on engineering analysis of static needs and operational environments. Aligning different cadences between systems engineering and software engineering activities may be challenging, but adjustments can maintain or improve either (or both) discipline's value. One significant clash is the impact of complex integrated master schedule planning in such detail and over such long time periods that the opportunity value of uncertainty collapses into engineering constraints and becomes a significant risk to success.

Principle 8: Unlock the Intrinsic Motivation of Knowledge Workers.

To ensure motivation and engagement among team members, create an environment marked by autonomy, mutual respect, and mission understanding.

Commentary: This principle likely does not affect most systems engineering technical activities. However, effectively managing the systems engineering workforce entails considering whether the software engineering and other disciplines sufficiently engage the systems engineering personnel to maintain interest and situational awareness. This principle is fundamental in large complex programs, such as weapons systems, highly regulated systems, and systems of systems, where the work spreads across numerous organizations or companies.

Principle 9: Decentralize Decision Making.

Decentralized decision-making is a key component for achieving the shortest sustainable value-delivery time.

Commentary: Decisions requiring sequential acceptance by multiple authority levels can destroy cadence, delay progress, and often lead to decisions based on outdated information. Strategic decisions are more effective if centralized, but teams should strive to delegate all other decisions to the level closest to the information involved and balance the need for continuous collaboration with the delays of sequential acceptance. Most systems engineering activities support rather than make decisions. Regardless of the decision-maker, those closest to the problem should develop the recommendations made by the systems engineering workforce. Those making a recommendation must have sufficient access to information and the visibility to understand the recommendation consequences. Analysis paralysis is contagious and should never become a factor (See variability and options above).

COMPARING DEVSECOPS PRINCIPLES TO SYSTEMS ENGINEERING

DevSecOps broadens these principles, and they help integrate development, security, and operation activities into a continuous integration/continuous deployment (CI/CD) pipeline (Wrubel and Yasar 2018). The SEI (Morales et al. 2020) defines these principles as follows:

Principle 1: Collaboration.

Full stakeholder engagement in every software development lifecycle aspect facilitates full awareness and input on all decisions and outcomes. Developers, operators, engineers, end-users, customers, and other stakeholders are active participants in decision-making and work progress (Davis 2017).

Commentary: Having ongoing access to systems engineering expertise is key in maintaining DevSecOps activities. Similarly, having software engineers involved in technical systems engineering activities reduces significant conflict and associated rework opportunities. Collaboration among systems and software engineers can also improve collaboration with project and program management.

Principle 2: Infrastructure as Code (IaC).

Klein (2018) explains IaC are software artifacts specifying the hardware/software components needed to run correctly and accessing, configuring, and installing each artifact. Infrastructure components can be actual, virtual, or both.

Commentary: While IaC is not specifically a systems engineering activity, IaC provides complete documentation of the execution environment maintained in the same repository as the code and supports the configuration management issues often plaguing software and system components. It also eases transitioning the code to an altered or completely new environment by providing a clear expectation description and identifying what software components need changing.

Principle 3: Continuous Integration.

Continuous integration automatically unifies individual system components into a single entity (Cois 2015). Unification occurs regularly, and the components, once unified, function together as a whole. The components may have dependencies on one another to function correctly.

Commentary: When coupled with IaC, continuous integration implements short learning cycles/increments giving systems engineering constant visibility into the code and ensures teams or teams of teams develop code while avoiding unexpected integration problems late in the development cycle. Rather than developing multiple components or capabilities in separate insular silos, continuous integration enables rapid access to integration issues before they cause significant rework (See the Environment Parity principle).

Principle 4 and 5: Continuous Delivery and Continuous Deployment.

Continuous delivery refers to automated software transfer to a staging environment similar to the production environment. Once delivered, the operations organization may conduct further testing but must decide when to deploy the software manually—for example, unclassified software running on classified data produced by another system, independently changing. Operations may want independent testing using live data before deployment. It also allows the operations team to decide if updates are valuable enough to deploy.

Continuous deployments need no operations team activity and transfer operational software directly into a production environment. It relies solely on rigorous static source code testing and dynamic deployable artifact testing within the CI/CD pipeline.

Commentary: Both continuous modes pass the fully integrated and tested software, including complete documentation and deployment information, to the operational organization. A continuous user transition mode provides a more rapid resolution for evolving cybersecurity vulnerabilities. While both methods reduce delay in capability delivery, each provides for different circumstances. When completing the testing in a duplicated operational environment, continuous deployment makes sense. If there is not absolute congruity between the testing environment and the operating environment—perhaps because of security or infrastructure needs—continuous delivery allows the organization to adjust the deployment cadence to their need without impacting the software development velocity. Continuous delivery/deployment provides systems engineering with a complete, fully documented software sequence. The drawbacks include the trust required and the rapid baseline evolution.

Principle 6: Environment Parity.

When two or more system environments are as identical as possible, they are in parity. DevSecOps pursues parity between development, staging, and production environments. IaC and deployable artifacts are critical to achieving parity.

Commentary: Like IaC, maintaining environment parity supports continuous testing integration and acceleration. An environment parity maintenance example is including security testing from the initial development through deployment. A constantly changing environment risks significantly delaying defect identification due to an environmental anomaly.

Principle 7: Automation.

A pipeline is technically implementing DevSecOps principles to assist all stakeholders in every software development aspect, including building, testing, delivery, and monitoring (Ficorilli 2017). Automated pipelines can also include configuration management and test environment generation.

Commentary: Automation significantly impacts systems engineering by providing substantial software status visibility, verification, and validation throughout the lifecycle (Nielsen 2019). It ensures teams perform testing at every level and do not sign off on any package until they integrate and test it. Automation also enables earlier and consistent verification and validation inclusion across systems and components.

Principle 8: Monitoring.

Continuous performance metric monitoring simultaneously drives pipeline improvement and software quality. Teams also need to monitor security for both the developing software and the pipeline automation.

Commentary: Historically, systems engineering monitoring focuses on key performance parameters and engineering targets. These are still critical tracking values. In a continuous integration and deployment environment, the information available for determining or demonstrating actual values will be more inclusive and more frequent. Systems engineers should have consistently better data to track

SO NOW WHAT?

It appears the principles generally align, but the foci of the practices are very different. Agile, lean, and DevSecOps are narrow, specific, and highly automated. Systems engineering practices incorporate the broader systems view. These should be mutually supportive. Unfortunately, many practice contexts, values, and incentives run counter to other practices within and between both disciplines. This is not insurmountable, but we need collaboration on mitigations and solutions. Hopefully, both disciplines strive to grow their understanding of the other's needs and goals, and the general principle alignment will provide room for innovation and improving outcomes. ■

ACKNOWLEDGMENTS

Copyright 2021 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM21-0296

REFERENCES

- Atlassian. 2019. "Kanban WIP Limits-Agile Coach 2019." *YouTube*, 3 April. https://www.youtube.com/watch?v=zEJn6eQO6FE.
- Boehm, B., D. Port, and M. Al-Said. 2020. "Avoiding the Software Model-Clash Spiderweb." *Computer* 33: 120-122. DOI:10.1109/2.881698
- Boehm, B., and R. Turner. 2015. "The Incremental Commitment Spiral Model (ICSM): Principles and Practices for Successful Systems and Software." Paper presented at the 2015 International Conference on Software and Systems Process, Tallinn, EE, 24-26 August.
- Centers for Disease Control and Prevention. 2017. "The Value of Systems Thinking." *YouTube*, 26 October. https://www.youtube.com/watch?v=Fo3ndxVOZEo.

- Cois, C. A. 2015. "Continuous Integration in DevOps." *Software Engineering Institute Blog*, 26 January. https://insights.sei.cmu.edu/blog/continuous-integration-in-devops/.
- Davis, C. 2017. "DevOps Who Does What-Cornelia Davis." *YouTube*, 12 June. https://www.youtube.com/watch?v=kpuJSp-p3hhA.
- Ficorilli, S. 2017. "Microcosm: A secure DevOps Pipeline as code." *Software Engineering Institute Blog*, 22 June. https://insights.sei.cmu.edu/blog/microcosm-a-secure-devops-pipeline-as-code/.
- Hayes, W. 2017. "Cadence in Agile Development." *YouTube*, 12 October. https://www.youtube.com/watch?v=UgXcOsmfVM8.
- Klein, J. 2018. "Infrastructure as Code: Moving Beyond DevOps and Agile." *Software Engineering Institue Blog*, 11 June. https://insights.sei.cmu.edu/blog/infrastructure-as-code-moving-beyond-devops-and-agile/.
- Matts, C. 2017. "DevOpsChat: Real Options Interview with Chris Matts." *YouTube*, 20 April. https://www.youtube.com/watch?v=YrEhH9R3NYg.
- Miller, S. 2019. "Agile Pitfall in Acquisition: The Bottom of the V." *Software Engineering Institute Blog*, 10 October. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=635239.
- Morales, J. A., R. Turner, S. Miller, P. Capell, P. R. Place, and D. J. Shepard. 2020. "Guide to Implementing DevSecOps for a System of Systems in Highly Regulated Environments." Technical Report, Carnegie Mellon University (Pittsburgh, US-PA).
- Nielsen, P. 2019. "The Modern Software Factory and Independent V&V for Machine Learning: Two Key Recommendations for Improving Software in Defense Systems." *Software Engineering Institute Blog*, 25 February. https://insights.sei.cmu.edu/blog/the-modern-software-factory-and-independent-vv-for-machine-learning-two-key-recommendations-for-improving-software-in-defense-systems/.
- Reinertsen, D. 2017. "4. Don Reinertsen: The Economics of Batch Size and the 'Father-Egg' Story." *YouTube*, 12 April. https://www.youtube.com/watch?v=zVASqSj_kvc.
- ———. 2020. "Let It Flow." *YouTube*, 24 March. https://www.youtube.com/watch?app=desktop&t=276&v=hTMwhjVVYu4&feature=youtu.be.
- Scaled Agile Framework (SAFe). 2020. "SAFe: Framework for Scaling Agile." https://www.scaledagile.com/enterprise-solutions/what-is-safe/.
- Wrubel, E., and H. Yasar. 2018. "Continuous Iterative Development and Deployment Practices." *Software Engineering Institute blog*, 22 October. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=528893

ADDITIONAL RESOURCES

- Boehm, B., J. A. Lane, S. Koolmanojwong, and R. Turner. 2013. *The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software*. Boston, US-MA: Addison-Wesley Professional.
- Boehm, B., and A. Jain. 2007. "The Value-Based Theory of Systems Engineering: Identifying and Explaining Dependencies." University of Southern California (Los Angeles, US-CA).
- Morales, J. 2019. "Challenges to Implementing DevOps in Highly Regulated Environments: First in a Series." *Software Engineering Institute Blog*, 29 January. https://insights.sei.cmu.edu/blog/challenges-to-implementing-devops-in-highly-regulated-environments-first-in-a-series/.
- Wrubel, E., S. Miller, M. A. Lapham, and T. A. Chick. 2014. "Agile Software Teams: How They Engage with Systems Engineering on DoD Acquisition Programs." Technical Note, Carnegie Mellon University (Pittsburgh, US-PA).

ABOUT THE AUTHOR

Richard Turner has over 40 years of systems, software, and acquisition engineering experience. He developed and acquired software in the private and public sectors and consulted for government and commercial organizations. Currently a Continuous Deployment of Capability Directorate member at Carnegie Mellon University's Software Engineering Institute, he recently spent 18 months working in the F-35 joint program office's modeling and simulation organization. His career includes nine years at the FAA and 10 years as faculty in the School of Systems and Enterprises and the Systems Engineering Research Center at Stevens Institute of Technology.

Dr. Turner's research interests include:

- harmonizing software, systems engineering, and acquisition lifecycle models
- applying agile and lean techniques to improve a wide variety of engineering tasks
- transitioning research to practice
- educating engineers in lean-agile principles and practice
- employing spiral, risk-driven methods in large system-of-systems acquisitions.

Dr. Turner is co-author of four books: *The Incremental Commitment Spiral Model: Principals and Practices for Successful Systems and Software* (Addison Wesley, 2014) and *Balancing Agility and Discipline: A Guide for the Perplexed* (Addison Wesley, 2004), both co-authored with Barry Boehm, *CMMISM Distilled* (Addison Wesley, 3rd edition 2008), and *CMMISM Survival Guide: Just enough Process Improvement* (Addison Wesley 2007), co-authored with Suzanne Garcia [Miller]. He was a core author for the IEEE-CS/PMI Software Extension to the *Guide to the PMBOK* and has received the Golden Core award from IEEE-CS.

Systems Thinking and Business Resilience: Questions That Should Keep Us Up at Night

Jeannine Siviy, jeannine.siviy@yahoo.com; and **Gene Kim,** genek@itrevolution.com

ABSTRACT
As interdisciplinary systems thinkers, we in the systems engineering field – especially the members of the Systems and Software Interface Working Group (SaSIWG) – are wired to believe that a "systems mindset should be everywhere." To catalyze the conversation about increasing the relevance of systems engineering and systems thinking in software- and data-intensive settings, this paper offers eight thought questions, which span governance, people, process, technology, and business concerns. The goal: resilience in the face of ever-increasing volatility and complexity. Join our conversation!

KEYWORDS: anti-fragility, complexity, resilience, systems engineering, systems thinking

INTRODUCTION AND BACKGROUND

"…Most of the problems faced by humankind concerns our inability to grasp and manage the increasingly complex systems of our world." —Peter Senge

We have become accustomed to the phrases "everything is software" and "everything is data" as well as "everything is being digitally transformed."

"Everything is a system" is not spoken with the same prevalence. Yet most things we encounter day to day are systems, as evidenced by these basic definitions:

- a set of things working together as parts of a mechanism or an interconnecting network. [Oxford languages]
- a group of interacting or interrelated entities [parts] that form a unified whole. [Wikipedia]
- a set of principles or procedures according to which something is done, an organized framework or method. [Oxford languages]
- an ordered and comprehensive assemblage of facts, principles, doctrines, or the like in a particular field of knowledge or thought: a system of philosophy. [Dictionary.com]
- a coordinated body of methods or a scheme or plan of procedure. [Dictionary.com]
- An *engineered system* is a system designed or adapted to interact with an operational environment to achieve one or more intended purposes while complying with applicable constraints. (INCOSE)

As an interdisciplinary field, systems engineering's bodies of knowledge are wide and deep, going well beyond a "product and technology" focus that many might expect. They span strategy, governance, people, process, and methods (including architecture, modeling, and analysis), and data.

With systems thinking principles at our core, we hold a variety of roles in organizations, we work to mission and we are wired to believe that a "systems mindset should be everywhere."

We observe that there are gaps in this mindset in many organizations. We seek to close these gaps, positioning organizations for not only flexibility and adaptability but also robustness, resilience, and anti-fragility[1] at speed and scale, in the face of ever-increasing volatility, uncertainty, complexity and ambiguity (aka VUCA).

If we are to realize this vision and increase the relevance (and usability) of systems engineering and systems thinking in software- and data-intensive organizations and programs, we need to apply our own expertise to the challenge: take a systems view, starting with key questions and leading to useful models (which we love to create!).

Eight questions, with elaborations, are shared here to catalyze the conversation.

QUESTIONS OF PEOPLE, PROCESS, TECHNOLOGY

"Systems Thinking is a discipline for seeing … complex situations, and for discerning high from low leverage change… By seeing wholes, we learn how to foster health." —Peter Senge

1 Anti-fragility is a property where systems capability increases in response to stressors, volatility, variability, or uncertainty. It differs from robustness (recovery from failure) and resilience (resistance to failure). (Taleb)

If as a matter of principle, we accept the value of an increased systems mindset, then we need to address fundamental questions of context (for situational awareness); roles; processes, tools, and methods; and people and their capabilities. By fostering this broad understanding, we can model effective ways to operationally integrate and interface systems, software, and data.

1. **Context:** How do we characterize and categorize software-, systems- and data-intensive settings? Consider:
   - types of systems, products and services produced: e.g., embedded real-time, Information Technology (IT) infrastructure, shrink-wrapped product with unpredictable behavior in sometimes unpredictable ecosystem behavior.
   - sectors and their classes: safety critical, like flight and health, finance, retail.
   - complexity: of the product, service, or system being produced.
   - more complexity: of the ecosystem or market in which it functions, such as regulated markets, and 2-sided platforms and markets.
   - size: number of people, revenue generated, and other ways to measure.

2. **Roles**: What roles should be held by systems thinkers throughout an organization and particularly relative to leadership, governance, decision-making and coaching accountabilities? Consider:
   - What should be the systems presence in executive roles as well as in program and engineering arenas?
   - Which roles are "all about systems" (e.g., "Chief Engineer" or "Chief Architect") and which roles need to be infused with systems thinking (e.g., C-Suite)?
   - How do we reconcile overlapping or duplicative role titles across disciplines? e.g., "architect" (preceded with any number of adjectives and qualifiers).
   - Is broad cross-industry consensus on this topic possible or practical?

3. **Processes and Methods:** How does systems engineering dovetail with software engineering, data science, and (new!) digital engineering in the execution of work? This is not a trivial question. A few notes on each are offered here to highlight connections (and hopefully will not draw ire because of incompleteness). Consider:
   - Systems engineering is an **interdisciplinary** field that focuses on how to design, integrate, and manage complex systems over their life. Its spans **products, services as well as enterprises**. With systems thinking at its core, it **unifies all disciplines** participating in an endeavor, often leveraging mathematical, graphical, and engineering **models**.
   - Software engineering is the application of computer science, software development, and related fields to build applications, operating **systems** and **systems** software that solve a broad range of **business problems**.
   - Data science is an **interdisciplinary** field at the intersection of software development, computer science, mathematics, data structures and business. Data scientists solve **complex problems** via **models** (preferably causal).
   - Digital Engineering is an **integrated** approach that uses authoritative sources of **system data and models** as a **continuum across disciplines** to support **lifecycle** activities from concept through disposal. (www.wpafb.af.mil)
   - Prevalent frameworks and life cycles, through which we all conduct work, include Agile, Scaled Agile, DevSecOps and Lean Portfolio Management.

4. **People:** How much "systems mindset" should software engineers and developers have? How much "software mindset" should systems engineers have? Consider:
   - In the context of Questions 2 and 3, and honoring each discipline's bodies of knowledge, how do we pinpoint specific necessary shared principles and skills?
   - Then how do we incorporate them into the state of the practice?

5. **Synthesis:** How do we, as systems engineers and systems thinkers, apply our own tools to create consumable, elegant and useful models for the preceding questions? Consider:
   - What we do is valuable, but often overwhelming to people, hence this dialogue.
   - What are our most basic tools, for clarifying interfaces, data flow, and cause and effect? When do we use those vs. more sophisticated methods?
   - What reusable models can we build? What must be handled within each organization?
   - How do we engage participants, one step at a time? How do we become masterful facilitators? e.g., group modeling using cause and effect loop diagrams.
   - And at the end of the day, remember the words of George Box: "All Models Are Wrong; Some Models Are Useful." We must strive for useful.

MAKING IT REAL: QUESTIONS OF BUSINESS

"94% of problems in business are systems driven…" —W. Edwards Deming

"Leaders: If you're too busy to build good systems, then you'll always be too busy." —Brian Logue

We will get the most traction on our vision and above tactical questions when there is good business fit, which brings us to a short list of complementary questions:

1. Fundamentally, what is the business case – the concrete business value – for implementing broad models of systems engineering and systems thinking throughout the organization? Consider:
   - How do we monetize matters of resilience and anti-fragility? Adaptability and flexibility? Speed of innovation?
   - How do we convince the constituencies that it matters to be proactive, and that systems thinking and engineering are key to success?

2. Culturally, how do we persuade organizational leaders to proactively "engineer the systems"? Consider
   - What is the scope and scale of "engineering the systems"? e.g., products, services, organizational systems (business and technical strategies, processes, culture, structure), interfaces and interactions with market and regulatory ecosystems
   - What do we mean by proactive vs. responsive vs. reactive?
   - Who should be accountable to understand, model, build and direct these systems?
   - What is the leadership and followership model? (Followership: having the courage of one's convictions to execute the vision of the leader within the framework of personal accountabilities (Geldart); to demonstrate teamwork, to build cohesion among the organization (Suda))

3. How do we stay abreast of industry trends (such as "digital engineering" which has been included in this paper) as well as within-organization trends, ensuring that the business, operating and technical models we set forth remain fresh and relevant? Consider:
   - How do we build effective and active coalitions with business and cross-discipline thought leaders? With our internal colleagues?
   - How does systems engineering have a "seat at the table" and be "in the conversation" relative to business, operational and technical strategy?

PLEASE JOIN US IN THE CONVERSATION

The time has come to mainstream systems engineering and systems thinking, as we strive to ensure that our software- and data-intensive systems and organizations are resilient and anti-fragile.

Systems engineering is both the object of this endeavor and the enabler of the solution. We systems engineering practitioners are masters of our fate, if you will. We thrive on navigating complexity to meet our mission. We have the wherewithal to create broad reusable models as well as specific fit-for-use models needed for each organization's journey.

With these questions and perspectives offered in this special issue, we begin to create agency for us to do so. We invite you to join the dialogue. ■

REFERENCES

- "Question 3, Processes and Methods" reflects author experience and several sources:
- (www.wpafb.af.mil) https://www.wpafb.af.mil/News/Article-Display/Article/2044411/digital-engineering-transformation-coming-to-the-af-weapons-enterprise/
- https://medium.com/@udemeudofia01/disciplines-in-data-science-a1da93306528
- https://thedatascientist.com/data-science-considered-own-discipline/
- https://en.wikipedia.org/wiki/Outline_of_software_engineering
- https://en.wikipedia.org/wiki/Systems_engineering
- (INCOSE) https://www.incose.org/about-systems-engineering/system-and-se-definition
- Each field has its own bodies of knowledge and references, well beyond these sources.
- (Geldart) Geldart, P. "Followership in Leadership," Eagles-Flight.com Blog, 10/28/20.
- (Suda) Suda, L., "In Praise of Followers," Project Management Institute Global Congress 2013.
- (Taleb) The definition of anti-fragility is drawn from Wikipedia summary of N. N. Taleb's work.

ABOUT THE AUTHORS

Jeannine Siviy is a business and technology strategist who recognizes undiscovered possibilities and spearheads paths of practical innovation – cutting through complexity and ambiguity, and delivering value at speed, at scale. She leads SDLC Partners' Healthcare Solutions, where she and her team passionately address persistent systems interoperation, automation, and ecosystem challenges, with one solution earning a Gartner Hype Cycle mention. She previously held leadership and technical roles at UPMC, Carnegie Mellon's Software Engineering Institute, and Eastman Kodak Company. She holds engineering degrees from Purdue and RIT, and Caltech's certificate in Technology & Innovation Management. A Pittsburgh native, she enjoys its cultural diversity and has a long-standing passion for nature and animals.

Gene Kim has been studying high performing technology organizations since 1999. He was founder and CTO of Tripwire, Inc for 13 years, an enterprise security software company. His books have sold over 1 million copies. He is author of the WSJ bestselling book *The Unicorn Project,* and was co-author of *The Phoenix Project*, *The DevOps Handbook*, and the Shingo Publication Award winning *Accelerate*. Since 2014, he has been the organizer of the DevOps Enterprise Summit, studying the technology transformations of large, complex organizations. He lives in Portland, Oregon with his wife and family.

White and Bouyard continued from page 31

REFERENCES

- Boardman, J., and B. Sauser. 2008. *Systems Thinking: Coping With 21st Century Problems*. Boca Raton, US-FL: CRC Press.
- Checkland, P. 1999. *Systems Thinking, Systems Practice—Soft Systems Methodology: A 30 Year Perspective*. New York, US-NY: Wiley.
- Creamer, N. 2021. "Voltaire." Goodreads, 17 February. Personal communication. https://www.goodreads.com/author/show/5754446.Voltaire.
- Kim, G. 2019. *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. Portland, US-OR: IT Revolution.
- Gide, A. 2021. "The Week." INews.co.uk.
- Perrow, C. 1999. *Normal Accidents: Living with High-Risk Technologies*. Princeton, US-NJ: Princeton University Press.
- Taleb, N. N. 2007. *The Black Swan*. New York, US-NY: Random House.
- ———. 2012. *Antifragile: Things that Gain from Disorder.* New York, US-NY: Random House.
- White, B. E. 2015. "On Leadership in the Complex Adaptive Systems Engineering of Enterprise Transformation." Journal of Enterprise Transformation 5 (3): 192-217. Supplementary Material (Appendices): http://www.tandfonline.com/doi/suppl/10.1080/19488289.2015.1056450.
- ———. 2021. *Toward Solving Complex Human Problems: Techniques for Increasing Our Understanding of What Matters in Doing So*. Boca Raton, US-FL: CRC Press.

ABOUT THE AUTHORS

Dr. Brian E. White received his Ph.D. and M.S. degrees in Computer Sciences from the University of Wisconsin and his S.M. and S.B. degrees in Electrical Engineering from M.I.T. He served in the US Air Force, and for eight years, was at M.I.T. Lincoln Laboratory. For five years, Dr. White was a principal engineering manager at Signatron, Inc. In his 28 years at The MITRE Corporation, he held various senior professional staff and project/resource management positions. He was MITRE's systems engineering process office director, 2003-2009. Dr. White retired from MITRE in July 2010, and has since offered a consulting service, CAU← SES ("Complexity Are Us" ← Systems Engineering Strategies). He taught as an adjunct professor at several US universities and currently tutors students in basic mathematics, calculus, electrical engineering, and complex systems. He edited and authored several published books and book chapters, mostly in his book series on complex and enterprise systems engineering with Taylor and Francis and the CRC Press. He presented a dozen tutorials in complex systems and published over one hundred conference papers and journal articles in complex systems, systems engineering, and digital communications over his 55+ year career.

Mickael Bouyaud is a business architect for Worldline, a global leader in seamless payments, and the technical director of AFIS, the French chapter of INCOSE. He has expertise in payment systems, specializing in deploying acceptance solutions in retail organizations, mobile security, a PIN on the Mobile solution, and an Android-based point of sale. He worked in the mobile industry as a 3GPP standard and algorithm engineer for Mitsubishi, then as a system architect for NXP and Ericsson.

Systems Engineering: The Journal of The International Council on Systems Engineering

Call for Papers

The *Systems Engineering* journal is intended to be a primary source of multidisciplinary information for the systems engineering and management of products and services, and processes of all types. Systems engineering activities involve the technologies and system management approaches needed for

- definition of systems, including identification of user requirements and technological specifications;
- development of systems, including conceptual architectures, tradeoff of design concepts, configuration management during system development, integration of new systems with legacy systems, integrated product and process development; and
- deployment of systems, including operational test and evaluation, maintenance over an extended life cycle, and re-engineering.

Systems Engineering is the archival journal of, and exists to serve the following objectives of, the International Council on Systems Engineering (INCOSE):

- To provide a focal point for dissemination of systems engineering knowledge
- To promote collaboration in systems engineering education and research
- To encourage and assure establishment of professional standards for integrity in the practice of systems engineering
- To improve the professional status of all those engaged in the practice of systems engineering
- To encourage governmental and industrial support for research and educational programs that will improve the systems engineering process and its practice

The journal supports these goals by providing a continuing, respected publication of peer-reviewed results from research and development in the area of systems engineering. Systems engineering is defined broadly in this context as an interdisciplinary approach and means to enable the realization of successful systems that are of high quality, cost-effective, and trustworthy in meeting customer requirements.

The *Systems Engineering* journal is dedicated to all aspects of the engineering of systems: technical, management, economic, and social. It focuses on the life cycle processes needed to create trustworthy and high-quality systems. It will also emphasize the systems management efforts needed to define, develop, and deploy trustworthy and high quality processes for the production of systems. Within this, *Systems Engineering* is especially concerned with evaluation of the efficiency and effectiveness of systems management, technical direction, and integration of systems. *Systems Engineering* is also very concerned with the engineering of systems that support sustainable development. Modern systems, including both products and services, are often very knowledge-intensive, and are found in both the public and private sectors. The journal emphasizes strategic and program management of these, and the information and knowledge base for knowledge principles, knowledge practices, and knowledge perspectives for the engineering of systems. Definitive case studies involving systems engineering practice are especially welcome.

The journal is a primary source of information for the systems engineering of products and services that are generally large in scale, scope, and complexity. *Systems Engineering* will be especially concerned with process- or product-line–related efforts needed to produce products that are trustworthy and of high quality, and that are cost effective in meeting user needs. A major component of this is system cost and operational effectiveness determination, and the development of processes that ensure that products are cost effective. This requires the integration of a number of engineering disciplines necessary for the definition, development, and deployment of complex systems. It also requires attention to the lifecycle process used to produce systems, and the integration of systems, including legacy systems, at various architectural levels. In addition, appropriate systems management of information and knowledge across technologies, organizations, and environments is also needed to insure a sustainable world.

The journal will accept and review submissions in English from any author, in any global locality, whether or not the author is an INCOSE member. A body of international peers will review all submissions, and the reviewers will suggest potential revisions to the author, with the intent to achieve published papers that

- relate to the field of systems engineering;
- represent new, previously unpublished work;
- advance the state of knowledge of the field; and
- conform to a high standard of scholarly presentation.

Editorial selection of works for publication will be made based on content, without regard to the stature of the authors. Selections will include a wide variety of international works, recognizing and supporting the essential breadth and universality of the field. Final selection of papers for publication, and the form of publication, shall rest with the editor.

Submission of quality papers for review is strongly encouraged. The review process is estimated to take three months, occasionally longer for hard-copy manuscript.

Systems Engineering operates an online submission and peer review system that allows authors to submit articles online and track their progress, throughout the peer-review process, via a web interface. All papers submitted to *Systems Engineering*, including revisions or resubmissions of prior manuscripts, must be made through the online system. Contributions sent through regular mail on paper or emails with attachments will not be reviewed or acknowledged.

All manuscripts must be submitted online to *Systems Engineering* at ScholarOne Manuscripts, located at:

http://mc.manuscriptcentral.com/SYS

Full instructions and support are available on the site, and a user ID and password can be obtained on the first visit.

INCOSE
Future events

Future events

JUL
17-22

31st Annual INCOSE International Symposium 2021
Virtual Event
www.incose.org/symp2021

SEP
17-19

2021 Western States Regional Conference (WSRC)
San Diego, CA, USA
www.incose.org/wsrc/wsrc2021

SEP
20-24

9th Nordic Systems Engineering Tour 2021
20th (Helsinki), 21st (Stockholm), 22nd (Oslo), 23rd (Copenhagen), 24th (Hamburg)
www.nordic-systems-engineering-tour.com

Early
OCT

New England Workshop 2021
Virtual event

OCT
28-29

EMEA Workshop 2021
Sevilla, Spain
www.incose.org/EMEAWS2021

NOV
17-19

INCOSE Human Systems Integration Conference 2021
San Diego, CA, USA
www.incose.org/HSI2021

JAN
29-Feb 1

Annual INCOSE International Workshop 2022
Torrance, CA, USA
www.incose.org/iw2022

JUNE
25-30

32nd Annual INCOSE International Symposium 2022
Detroit, MI, USA
www.incose.org/symp2022

INCOSE

HSI2021
Human Systems
Integration
Conference

San Diego, CA, USA

Save the date

November 17-19, 2021
www.incose.org/hsi2021