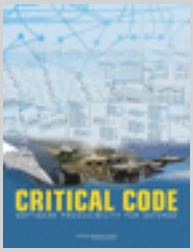

Critical Code: Software Producibility for Defense

ISBN
978-0-309-15948-7

160 pages
8 1/2 x 11
PAPERBACK (2010)

Committee for Advancing Software-Intensive Systems Producibility; National Research Council

CRITICAL CODE

SOFTWARE PRODUCIBILITY FOR DEFENSE

Committee for Advancing Software-Intensive Systems Producibility

Computer Science and Telecommunications Board

Division on Engineering and Physical Sciences

NATIONAL RESEARCH COUNCIL
OF THE NATIONAL ACADEMIES

THE NATIONAL ACADEMIES PRESS
Washington, D.C.
www.nap.edu

THE NATIONAL ACADEMIES PRESS **500 Fifth Street, N.W.** **Washington, DC 20001**

NOTICE: The project that is the subject of this report was approved by the Governing Board of the National Research Council, whose members are drawn from the councils of the National Academy of Sciences, the National Academy of Engineering, and the Institute of Medicine. The members of the committee responsible for the report were chosen for their special competences and with regard for appropriate balance.

Support for this project was provided by the Office of the Secretary of Defense, Department of Defense, with assistance from the National Science Foundation under sponsor award number CNS-0541636 and by the Office of Naval Research under sponsor award number N00014-04-1-0736. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the agencies and organizations that provided support for the project.

International Standard Book Number-13: 978-0-309-15948-7
International Standard Book Number-10: 0-309-15948-2

Additional copies of this report are available from the National Academies Press, 500 Fifth Street, N.W., Lockbox 285, Washington, DC 20055; (800) 624-6242 or (202) 334-3313 (in the Washington metropolitan area); Internet, http://www.nap.edu.

Printed in the United States of America

THE NATIONAL ACADEMIES

Advisers to the Nation on Science, Engineering, and Medicine

The **National Academy of Sciences** is a private, nonprofit, self-perpetuating society of distinguished scholars engaged in scientific and engineering research, dedicated to the furtherance of science and technology and to their use for the general welfare. Upon the authority of the charter granted to it by the Congress in 1863, the Academy has a mandate that requires it to advise the federal government on scientific and technical matters. Dr. Ralph J. Cicerone is president of the National Academy of Sciences.

The **National Academy of Engineering** was established in 1964, under the charter of the National Academy of Sciences, as a parallel organization of outstanding engineers. It is autonomous in its administration and in the selection of its members, sharing with the National Academy of Sciences the responsibility for advising the federal government. The National Academy of Engineering also sponsors engineering programs aimed at meeting national needs, encourages education and research, and recognizes the superior achievements of engineers. Dr. Charles M. Vest is president of the National Academy of Engineering.

The **Institute of Medicine** was established in 1970 by the National Academy of Sciences to secure the services of eminent members of appropriate professions in the examination of policy matters pertaining to the health of the public. The Institute acts under the responsibility given to the National Academy of Sciences by its congressional charter to be an adviser to the federal government and, upon its own initiative, to identify issues of medical care, research, and education. Dr. Harvey V. Fineberg is president of the Institute of Medicine.

The **National Research Council** was organized by the National Academy of Sciences in 1916 to associate the broad community of science and technology with the Academy's purposes of furthering knowledge and advising the federal government. Functioning in accordance with general policies determined by the Academy, the Council has become the principal operating agency of both the National Academy of Sciences and the National Academy of Engineering in providing services to the government, the public, and the scientific and engineering communities. The Council is administered jointly by both Academies and the Institute of Medicine. Dr. Ralph J. Cicerone and Dr. Charles M. Vest are chair and vice chair, respectively, of the National Research Council.

www.national-academies.org

COMMITTEE FOR ADVANCING SOFTWARE-INTENSIVE SYSTEMS PRODUCIBILITY

WILLIAM L. SCHERLIS, Carnegie Mellon University, *Chair*
ROBERT F. BEHLER, The MITRE Corporation
BARRY W. BOEHM, University of Southern California
LORI A. CLARKE, University of Massachusetts, Amherst
MICHAEL A. CUSUMANO, Massachusetts Institute of Technology
MARY ANN DAVIDSON, Oracle Corporation
LARRY DRUFFEL, Software Engineering Institute
RUSSELL FREW, Lockheed Martin
JAMES LARUS, Microsoft Corporation
GREG MORRISETT, Harvard University
WALKER ROYCE, IBM
DOUGLAS C. SCHMIDT, Carnegie Mellon University
JOHN P. STENBIT, Independent Consultant
KEVIN J. SULLIVAN, University of Virginia

Staff

JON EISENBERG, Director, CSTB
LYNETTE I. MILLETT, Senior Program Officer
JOAN D. WINSTON, Program Officer (until May 2008)
ENITA A. WILLIAMS, Associate Program Officer
ERIC WHITAKER, Senior Program Assistant

Preface

The Committee for Advancing Software-Intensive Systems Producibility was appointed by the National Research Council (NRC) and convened under the auspices of the NRC's Computer Science and Telecommunications Board (CSTB) to assess the nature of the national investment in software research and, in particular, to consider ways to revitalize the knowledge base needed to design, produce, and employ software-intensive systems for tomorrow's defense needs. The statement of task is provided in Box P.1.

This report contemplates Department of Defense (DoD) needs and priorities for software research and suggests a research agenda and related actions. This is the final report of the committee, and it builds on two prior reports—*Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*[1] and *Preliminary Observations on DoD Software Research Needs and Priorities*.[2] This report draws on the briefings listed in Appendix A.

The committee considered four sets of questions:

- To what extent is software capability significant for the DoD? Is it becoming more or less significant and strategic in systems development?
- Will the advances in software producibility needed by the DoD emerge unaided from industry at a pace sufficient to meet evolving defense requirements?
- What are the opportunities for the DoD to make more effective use of emerging technology to improve software capability and software producibility?
- In which technology areas should the DoD invest in research to advance defense software capability and producibility?

Chapter 1 of this report addresses the first two of these questions. It discusses the essential and evolving role of software in defense systems and the distinctive and unusual characteristics of the software

[1] National Research Council (NRC), 2007, *Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 10, 2010.

[2] NRC, 2008, *Preliminary Observations on DoD Software Research Needs and Priorities: A Letter Report*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12172. Last accessed August 10, 2010

BOX P.1
Statement of Task

This study will bring together academic and industry software systems researchers, software and software tool vendors (suppliers), and systems integrators who comprise the community of skills required for future successes in complex software-intensive systems required by the Department of Defense (DoD). They will:

(1) Assess the emerging situation with respect to the national investment in relevant software research, the present state of and future requirements for tools for software production, testing and maintenance, and the adequacy of human resources;

(2) Examine the needs, relationships, and interdependencies expected of future DoD software research, development and maintenance needs, and consider what advances are needed for continuous improvements in the design, production, and evolution of DoD software-intensive systems;

(3) Make recommendations to responsible agency, executive branch, and legislative officials, and to the software technical community, about how to improve the present state of affairs and achieve future goals.

used in such systems. The chapter also contemplates the extent to which the DoD can rely on industry to innovate at a rate fast enough to allow it to fully meet future defense software requirements.

Chapters 2, 3, and 4 of this report focus on three principal clusters of challenges to software producibility wherein the DoD has particularly unusual needs or "leading demand." These chapters address the third question presented in the statement of task and describe process management for innovative software systems development (Chapter 2), architectural leadership for large-scale software-intensive systems (Chapter 3), and the need to take a strategic approach to assurance (Chapter 4). These chapters, taken together, address the core features of what we mean by *software producibility*—the capacity to design, produce, assure, and evolve software-intensive systems in a predictable manner while effectively managing risk, cost, schedule, quality, and complexity.

Chapter 5 discusses the value of research in enhancing software producibility for the DoD. It addresses the role of academic research, the synergy between industry and academic research, and the impact of past investments. It then tackles the fourth question and offers a seven-part agenda for advancing DoD software capability: architecture, assurance, process and economic models, requirements, language and tools, cyber-physical systems, and human-systems interaction.

The committee thanks all those who participated in its workshops and contributed to its deliberations (Appendix A). The committee would also like to thank the Computer Science and Telecommunications Board staff, including Enita Williams, Jon Eisenberg, Lynette Millett, Joan Winston, and Eric Whitaker, who have ably managed the project and coordinated the team effort through three separate reports. Enita Williams and Jon Eisenberg deserve special thanks and appreciation for their heroic effort in the preparation and editing of this final report, which would not have been possible without their highly capable support and collaboration.

William L. Scherlis, *Chair*
Committee for Advancing Software-Intensive Systems Producibility

Acknowledgment of Reviewers

This report has been reviewed in draft form by individuals chosen for their diverse perspectives and technical expertise, in accordance with procedures approved by the National Research Council's (NRC's) Report Review Committee. The purpose of this independent review is to provide candid and critical comments that will assist the institution in making its published report as sound as possible and to ensure that the report meets institutional standards for objectivity, evidence, and responsiveness to the study charge. The review comments and draft manuscript remain confidential to protect the integrity of the deliberative process. We wish to thank the following individuals for their review of this report:

Rick Buskens, Lockheed Martin Advanced Technology Laboratories
Grady Campbell, Software Engineering Institute
William Campbell, BAE Systems
John Gilligan, Gilligan Group
William Griswold, University of California, San Diego
Anita Jones, University of Virginia
Annette Krygiel, Independent Consultant
Steve Lipner, Microsoft, Inc.
David Notkin, University of Washington
Frank Perry, SAIC
Alfred Z. Spector, Google, Inc.
Daniel C. Sturman, Google, Inc.
John Swainson, CA, Inc.
Mark N. Wegman, IBM

Although the reviewers listed above have provided many constructive comments and suggestions, they were not asked to endorse the conclusions or recommendations, nor did they see the final draft of the report before its release. The review of this report was overseen by William H. Press, University of Texas at Austin. Appointed by the NRC, he was responsible for making certain that an independent examination of this report was carried out in accordance with institutional procedures and that all review comments were carefully considered. Responsibility for the final content of this report rests entirely with the authoring committee and the institution.

Contents

Summary

The National Research Council's Committee for Advancing Software-Intensive Systems Producibility was commissioned by the Office of the Secretary of Defense (OSD) to examine the nature of the national investment in software research and ways to revitalize the knowledge base needed to design, produce, and employ software-intensive systems for tomorrow's defense needs. This report contemplates *Department of Defense needs and priorities* (Chapter 1) for software producibility—that is, the capacity to design, produce, assure, and evolve innovative software-intensive systems in a predictable manner while effectively managing risk, cost, schedule, and complexity. It suggests feasible actions related to *software process and measurement* (Chapter 2), *architecture* (Chapter 3), and *assurance* (Chapter 4), and it suggests a *research agenda* (Chapter 5) that focuses on issues critical to Department of Defense (DoD) software capability.

Box S.1 summarizes several of the key messages of the findings and recommendations by showing how they address eight "myths" regarding software producibility. The key findings and recommendations of the committee are presented in this Summary, and additional findings and recommendations are offered in subsequent chapters. A complete set is presented in Box S.2.

This final project report builds on two prior reports—the discussion of technical and organizational issues in *Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*[1] and a subsequent letter report focused on the rationale for investment in software research.[2]

[1] National Research Council (NRC), 2007, *Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 10, 2010.

[2] NRC, 2008, *Preliminary Observations on DoD Software Research Needs and Priorities: A Letter Report*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12172. Last accessed August 10, 2010

BOX S.1
Eight Myths About Defense Software Producibility

1. The DoD's software producibility challenges are predominantly challenges of *management* and *process* but not of *technology*.
   - (See Findings 1-1, 1-3, 1-4, 2-5, 4-2, 5-2 and Recommendations 1-1, 4-2, 5-1.)
2. The DoD and its contractors can rely on industry to innovate at a rate fast enough to solve the DoD's hard technical problems and to stay ahead of its adversaries.
   - (See Findings 1-3, 1-4 and Recommendation 1-1.)
3. Software technology is approaching a plateau, which diminishes the need to invest in technology innovation.
   - (See Findings 1-5, 5-2 and Recommendations 4-2, 5-1.)
4. The software research community is doing potentially relevant theoretical work, but it has not led to advances of compelling importance to the DoD.
   - (See Finding 5-1.)
5. We have not yet developed effective mechanisms to mitigate the risks, particularly those related to scale and adoptability, associated with the transition to practice of innovative software-development technologies.
   - (See Findings 3-2, 3-4, 3-5, 4-2, 4-3 and Recommendations 2-1, 3-4, 4-2, 4-3.)
6. We will never create perfectly reliable and secure software, so we should focus primarily on provenance—trusted sources—rather than attempting to achieve assurance through improvements in practices and tools for evaluating artifacts directly.
   - (See Findings 4-1, 4-2 and Recommendations 4-1, 4-3.)
7. There is sufficient software research already underway, sponsored primarily by NSF and other basic science agencies, to meet the DoD's software needs.
   - (See Recommendations 1-1, 5-1.)
8. Earned value management approaches based on code accumulation are a sufficient basis for managing software development programs, including incremental iterative development.
   - (See Findings 2-3, 2-4 and Recommendations 2-1, 2-2.)

1. RECOGNIZE THE PIVOTAL ROLE OF DOD SOFTWARE INNOVATION

The continued increase in the DoD's dependency on software is well documented by the Defense Science Board (DSB) and in multiple National Academies reports.[3,4,5,6] This increase amounts to an order of magnitude of lines of software code every decade, and it is a natural consequence of the distinctive advantages of software as an engineering medium. Software is uniquely unbounded and flexible, can

[3] Defense Science Board (DSB), September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[4] DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA385923. Last accessed August 20, 2010.

[5] National Research Council (NRC), 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 20, 2010.

[6] NRC, 1999, *Realizing the Potential of C4I: Fundamental Challenges*,Washington, DC: National Academy Press. Available online at http://www.nap.edu/catalog.php?record_id=6457. Last accessed August 20, 2010.

be delivered and upgraded electronically and remotely, and has the potential to be rapidly adapted to changing threats, operating environments, and platform technologies. Because it is unconstrained by traditional physical engineering limitations, the principal limits on what we can accomplish with software derive from human intellectual capacity to conceptualize and understand systems, to build tools to develop and manage them, and to provide assurance regarding critical functional and quality attributes.

The same reports also indicate that the DoD would benefit from strategic steps to improve its ability to design, develop, and assure complex software. Software not only is expanding in use but also is shifting into a more strategic and fundamental role in diverse systems. A vital question is how the DoD can ensure that it will be able to meet its software needs now and into the future.

Finding 1-1: Software has become essential to a vast range of military system capabilities and operations, and its role is continuing to deepen and broaden, including interlinking diverse system elements. This creates both benefits and risks.

Compounding these issues are the growing size, complexity, and geography of the supply chain structure for major software systems. This is a consequence of two powerful forces—the advance of technology that has enabled greater software modularization, and the globalization of software development activity. Although the United States continues to retain innovation leadership in software areas important to the DoD, there are factors that could cause the loss of that leadership.

Some observers have speculated that software and information technology generally are reaching a plateau of capability and performance. This is a false and dangerous speculation—the capability and the complexity of hardware[7] and software systems are both rising at an accelerating rate.

Finding 1-5: It is dangerous to conclude that we are reaching a plateau in capability and technology for software producibility. To avoid loss of leadership, the DoD will need to become more fully engaged in the innovative processes related to software producibility.

A key question addressed by the committee is to what extent the DoD, without providing any explicit R&D stimulus, can rely on industry—specifically the domestic defense industrial base and supporting vendors—to produce software innovations in areas of defense significance at a rate fast enough to allow the DoD to fully meet software requirements and remain ahead of potential adversaries. Finding the answer to this question is made more urgent by the expected continued rapid evolution of software capability worldwide. A loss of leadership could threaten the ability of the DoD not only to manifest world-leading capability, but also to achieve adequate levels of assurance for the diversely sourced software it intends to deploy. It will thus be essential for the DoD to reengage directly in the innovation process if it is to retain this necessary leadership. (See also Recommendation 5-1.)

Finding 1-4: The DoD's needs will not be sufficiently met through a combination of demand-pull from the military and technology-push from the defense or commercial information technology sectors. The DoD cannot rely on industry alone to address the long-term software challenges particular to defense.

Defense requirements for software are in many respects similar to requirements in other sectors. But there are important areas where the DoD must push the envelope beyond mainstream capability

[7] Moore's Law is an informal predictive model created by Gordon Moore in 1965 for the number of transistors on integrated circuit chips. For decades, there has been a close correlation of transistor count with both processor clock speeds and overall computing capacity. Recently, due to a combination of factors, clock speeds have leveled off or even diminished, while the growth in general-purpose computing capacity has been achieved through the provisioning of multiple processors (called "cores"). This has created an added challenge related to concurrency for software developers, as elaborated in Chapters 4 and 5.

in order to meet its mission needs. These areas of "leading demand" include, for example, software assurance in the presence of highly sophisticated adversaries, architectural innovation and complexity, criticality with respect to safety, overall complexity and scale, and the arm's-length relationship that the DoD has with its development teams—where mission stakeholders are often required to engage with development teams only through a legal and contractual interface.

Recommendation 1-1: The DoD, through its Director of Research and Engineering (DDR&E), should regularly undertake an identification of areas of technological need related to software producibility where the DoD has "leading demand" and where accelerated progress is needed to support the defense mission.

2. ACCEPT UNCERTAINTY: ATTACK RISKS AND EXPLOIT OPPORTUNITIES

The management of innovative software development is largely a process of managing risks. Experience shows that, in the absence of advanced process models, there is a correlation between the degree of precedent and routinization, on the one hand, and the ability to deliver results with predictable cost, schedule, and success in acceptance evaluation, on the other.

With regard to the precedented elements—whose users can benefit, in terms of design costs and risks, from the experience of existing and prior users—the DoD benefits by adjusting its practices to conform to government and industry conventions, enabling it to exploit a broader array of more mature market offerings. When applied to innovative systems, however, the familiar sequential ("waterfall") processes can often lead to costly surprises and increased programmatic risk. That is, what appears to be a "safe" conservative decision to follow the most basic process is in fact a dangerous decision that can drastically increase programmatic risk and the possibility of total project failure. The largest producibility challenges for the DoD, therefore, arise from its need to develop innovative, unprecedented software systems. Such efforts at development necessarily build on precedented elements, and the unprecedented aspects may create substantial programmatic risk unless managed effectively. Effective management means identifying and mitigating the engineering risks that derive primarily from the innovative elements—architecture, assurance, requirements, design, scale, performance, etc. A well-managed incremental and iterative process, supported by appropriate iterative evaluation and measurement approaches, can more reliably lead to successful outcomes—lowering programmatic risk, even when there are significant engineering risks.

Modern software governance is about managing uncertainty. This means treating project scope, plans, and resources as variables (not frozen baselines) and explicitly managing the variances in these variables until they converge to acceptable levels. This requires honest and well-informed assessments of engineering risks to effectively trade off cost, schedule, overall programmatic risk, and functionality.

When there is substantial software-manifest functionality as well as software-related risks, there should be a close coupling of design and process decisions relating to hardware, software, and human-systems integration, with prioritization based on identified criteria.[8]

Finding 2-1: Modern practice for innovative software systems at all levels of scale is geared toward incremental identification and mitigation of engineering uncertainties, including requirements uncertainties. For defense software, the challenge is doing so at a larger scale and in ways that are closely linked with an overall systems engineering process.

Following the practice of other organizations that manage large engineering projects, the DoD has

[8] Fred Brooks, 2010, *The Design of Design: Essays from a Computer Scientist*, Boston: Addison-Wesley. See also NRC, Richard Pew and Anne Mavor, eds., 2007, *Human-System Integration in the System Development Process: A New Look*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11893. Last accessed August 20, 2010.

adopted earned value management (EVM), which is "a means of determining the financial health of a project by measuring whether the work completed to date is in line with the budget and schedule planning." One of the reasons for using EVM is to get early warning of potential problems. EVM tracks plans, progress, cost, earned value (the *planned* cost of *actual* progress), and variances in cost and schedule. The underlying technique is seemingly straightforward, but the application of EVM to innovative and unprecedented software-intensive systems poses challenges. These derive from the choice of EVM assessment and measurement strategies. Significant improvements are needed in our ability to value the creation of software assets such as validated architecture and design commitments or evidence in support of quality assurances.

Finding 2-3: Extensions to earned value management models to include evidence of feasibility and to accommodate practices such as time-certain development are necessary conditions to enable successful application of incremental development practices for innovative systems.

Finding 2-4: Research related to process, measurement, architecture, and assurance can contribute to the improvement of measurement practice in support of both routine management of engineering risks and value assessment as part of earned value management.

The committee focuses (in Chapter 2) on six areas for improvement in the management of innovative software projects: (1) improved measurement and associated technology, (2) architecture validation using models, simulation, prototyping, etc., (3) program manager training and perceived career risks, (4) accretion of an accessible experience base and other shared resources that can facilitate sound decision making over the long term, (5) acceptable shifts of early-stage emphasis for innovative systems from detailed functional requirements to architecture, scope, and process definition, and (6) the need for flexibility and adaptation in long-lived projects.

Recommendation 2-1: The DoD should take aggressive actions to identify and remove barriers to the broader adoption of incremental development methods, including iterative approaches, staged acquisition, evidence-based systems and software engineering, and related methods that involve explicit acknowledgment and mitigation of engineering risk.

An additional difficulty is the lack of a common basis for judging cost estimates. There are well-used metrics for hardware, but a uniform set of standards for measurement in software development is lacking, although there are candidate models.

Recommendation 2-2: The DoD should take steps to accumulate high-quality data regarding project management experience and technology choices that can be used to inform cost estimation models, particularly as they apply to innovative software development projects.

It is widely acknowledged, including within the DoD, that the department does not have sufficient organic personnel with the software expertise to meet its needs for today's more software-intensive programs. This includes the expertise to effectively purchase the larger and less precedented systems as well as the precedented systems for which sensitivity to issues such as the choice of ecosystem is key. The necessary expertise includes understanding of process, architecture, requirements, and assurance, as well as of the trajectories and adoption trends for both the major commercial ecosystems and any involved DoD-intrinsic software ecosystems. Because the DoD does not currently have the requisite expertise and talent it needs for effective software producibility and the rapid pace of software development demands ongoing interaction with the field, the DoD must engage experts outside of the DoD and its primes. The DoD should adapt processes to facilitate input from outside experts throughout the systems-engineering lifecycle for software-intensive systems.

Finding 2-6: The DoD has a growing need for software expertise, and it is not able to meet this need through intrinsic resources. Nor is it able to fully outsource this requirement to DoD primes. The DoD needs to be a smart software customer. This need is particularly significant for large-scale innovative software-intensive projects for which there are cross-cutting software architectural requirements and validation challenges.

3. ASSERT DOD ARCHITECTURAL LEADERSHIP

The increasing complexity and scale of innovative software systems demand that the DoD play an active role in the definition of systems and software architecture throughout the project lifecycle. Software architecture is conventionally defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them."[9] Architecture is significant because it represents the earliest and often most important design decisions: those that are the hardest to change and the most critical to get right. Architecture is the first design artifact that addresses quality attributes such as performance, modifiability, reliability, security, and safety. Although having a well-matched architecture is not a guarantee of success, software systems that are not based on well-formulated software architectures are, in the committee's view, more likely to exhibit the kind of software horror stories too often experienced in DoD acquisitions with respect to project risk.

Finding 3-5: In systems with innovative functional or quality requirements, benefit is derived from an early focus on the most essential architectural commitments and quality attributes, with deferred commitment to specifics of functional characteristics. This approach can reduce the overall uncertainty of the engineering process and yield better outcomes.

Architectural decision making for any particular software development project is profoundly influenced by precedent—knowledge of related ecosystems, of systems and hardware infrastructure, of available frameworks and libraries, and of previous experience with similar systems and projects. Small changes to architectural requirements can open or close opportunities to exploit rich, existing ecosystems, greatly influencing both cost and risk.[10,11]

Finding 3-1: Industry leaders attend to software architecture as a first-order decision, and many follow a product-line strategy based on commitment to the most essential common software architectural elements and ecosystem structures.

Architecture embodies planning for flexibility—architecture commitments effectively define and encapsulate areas where change or diversity is anticipated, or not. Software architecture commitments thus enable product-line strategies.

Finding 3-2: The technology for definition and management of software architecture is sufficiently mature, with widespread adoption in industry. These approaches are ready for adoption by the DoD, assuming that a framework of incentives can be created in acquisition and development efforts.

The DoD experience with long-term software acquisition programs has provided strong evidence for the value of software architecture,[12] and there are examples of programs that have followed an archi-

[9] Len Bass, Paul Clements, and Rick Kazman, 2003, *Software Architecture in Practice*, 2nd Ed., Boston: Addison-Wesley.

[10] Dennis M. Buede, 2000, *The Engineering Design of Systems: Models and Methods,* New York: John Wiley & Sons, Inc., pp. 7-8, 25.

[11] Barry Boehm, Ricardo Valerdi, and Eric Honour, 2008, "The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems," *Systems Engineering* 11(3):221-234.

[12] Walker E. Royce, 1998, *Software Project Management: A Unified Framework*, Reading, MA: Addison-Wesley.

tecture-driven acquisition strategy. These illustrate the benefits of pervasive commitment to an architecture-driven approach[13,14] — including reduced engineering risk, reduced development and maintenance costs, decreased time to field, increased system agility, and improved system quality. The opportunity exists for the DoD to assert leadership across its diverse software-intensive systems portfolio.

It may be difficult to ascertain which kinds of architectural commitments are essential to an innovative project—at the outset of a project, a small number of well-crafted "seed" commitments may be sufficient to enable a direction to be set. Generally speaking, architecture in the early stages of an innovative project should be the minimum commitment that yields the maximum value with respect to quality attributes and capability to incrementally implement functional capabilities. Refinement and elaboration—further architectural commitment— is then undertaken as part of an incremental iterative process. A corollary of this approach is that architecture leadership is best undertaken by individuals engaged directly in the engineering process and is best separate from activities related to ecosystems certification and other standards-related policy setting.

Recommendation 3-2: This committee reiterates the past Defense Science Board recommendations that the DoD follow an architecture-driven acquisition strategy, and, where appropriate, use the software architecture as the basis for a product-line approach and for larger-scale systems potentially involving multiple lead contractors.

Recommendation 3-3: The DoD should enhance existing practices to afford better distinctions between precedented portions of systems and innovative portions of systems, wherein architectures are developed both to encapsulate the innovative elements and to afford maximum opportunity to build on experience and existing ecosystems for precedented elements. These overall architectures, and particularly the innovative elements, should be subject to early and continuous validation, especially in systems that have requirements for interoperation.

4. ADOPT A STRATEGIC APPROACH TO SOFTWARE ASSURANCE

One of the great challenges for both defense and civilian systems is software quality assurance. Software assurance encompasses reliability, security, robustness, safety, and other quality-related attributes as well as functionality and performance. Diverse studies suggest that overall software assurance costs account for 30 to 50 percent of total project costs for most software projects.[15] Despite this cost, current approaches to software assurance, primarily testing and inspection, are generally regarded as

[13] Mark Kasunic, 2004, *Army Strategic Software Improvement Program (ASSIP) Survey of Army Acquisition Managers, Technical Report*, Carnegie Mellon University/Software Engineering Institute (SEI), CMU/SEI-2004-TR-003. Available online at http://www.sei.cmu.edu/library/abstracts/reports/04tr003.cfm. Last accessed August 20, 2010.

[14] Peter H. Feiler and Dionisio de Niz, 2008, *ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System, Engineering Practices, Special Report*, Carnegie Mellon University/SEI, CMU/SEI-2008-SR-001. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA480129. Last accessed August 20, 2010.

[15] In "Software Debugging, Testing, and Verification" (*IBM Systems Journal* (41)1, 2002), Brent T. Hailpern and P. Santhanam say, "In a typical commercial development organization, the cost of providing this assurance via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost." In *Estimating Software Costs* (McGraw-Hill, 1998), T. Capers Jones provides a table relating percentage of defects removed vs. percentage of development effort devoted to testing, with data points, including 90 vs. 39, 96 vs. 48, and 99.9 vs. 58. In *Software Cost Estimation with COCOMO II* (Prentice Hall, 2000), Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece indicate that the cost of test planning and running tests is typically 20 to 30 percent plus rework due to defects discovered. In *Balancing Agility and Discipline: A Guide for the Perplexed* (Addison-Wesley, 2004), Barry Boehm and Richard Turner provide an analysis of the COCOMO II Architecture and Risk Resolution scale factor, indicating that the increase in rework due to poor architecture and risk resolution is roughly 18 percent for typical 10-KSLOC (KSLOC stands for thousand software lines of code) projects and roughly 91 percent for typical 10,000-KSLOC projects. (COCOMO II, or constructive cost model II, is a software cost, effort, and schedule estimation model.) This analysis suggests that improvements are needed in upfront areas as well as in testing and supporting the importance of architecture research, especially for ultra-large systems.

inadequate. Testing, for example, cannot yield assurance for many kinds of failures related to security or non-determinism.

In defense programs, the assurance process, including particularly the use of preventive approaches, is heavily complicated by the arms-length relationship that exists between a contractor development team and government stakeholders. Additionally, although the DoD relies extensively on vendor software and undertakes considerable testing of that software, it also implicitly relies on relationships founded in trust (rather than verification) to assure many quality attributes.[16]

Failures in software assurance can be of particularly high consequence for defense systems due to their growing roles in protecting human lives, in war fighting, and in safeguarding national assets. In many life and death situations, optimum performance may not be the proper overriding assurance criterion, but rather the "minimization of maximum regret." This is exacerbated by the fact that a full-scale operational test of many capabilities is not feasible, but assurance must nonetheless be achieved.

Software assurance is a human judgment of fitness for use. For defense systems, there is particular emphasis on addressing hazards related to security, availability and responsiveness, safety, policy adherence, and diverse other attributes, but there are many other quality attributes encompassed by software assurance. In practice, assurance judgments are based on application of a broad range of techniques that include both preventive and evaluative methods and that are applied throughout a software engineering process. It is false to conclude that assurance can be achieved entirely through acceptance evaluation such as achieved through DoD's operational and systems test processes. In particular, it is well understood by software engineers and managers that quality, including security, is not "tested in," but rather is "built in." But there are great challenges to succeeding both in building in quality (preventive methods) and in assuring that it is there (evaluative methods). From a process perspective, there is overlap between preventive and evaluative methods—when used at the earliest stages in the process, evaluative methods shorten feedback loops and guide development choices.

Development practices and technologies can profoundly influence the ability to achieve successful and cost-effective evaluation outcomes. These development choices range from choices of architecture to choices of programming language, coding style, and associated tooling. One of the great benefits of modern tooling is that a much more comprehensive record of development can be used to facilitate evaluation.

Software assurance is different from reliability analysis for physical systems. Unlike other engineering materials, software does not wear out or suffer transient faults. But it can suffer transient *errors*, for example, because of concurrency. This is both an obvious and a subtle point. It is obvious in the sense that there is no analog of metal fatigue, rust and oxidation, or other kinds of physical deterioration or environmentally induced change in physical properties. It is subtle because software is often the mechanism of choice for handling such faults in associated hardware. When software delivers bad results, including transient errors, they are due to permanently faulty software design, which must be addressed by changes in the software code.

The goal of assurance methods is to ultimately connect the code that is executed with architectural, functional, and quality requirements. Although software code is all that is necessary for the software to operate, considerable additional information is needed to effectively support ongoing evolution of the software over its lifespan, including architecture models, designs, test cases, etc. This information supports an incremental process, in which chains of evidence can be created with links among the artifacts being created (and adapted) as the development process proceeds. Validation of these traceability links comes from diverse techniques including testing, inspection, analysis, model checking, and simulation. An example of a link is a test case that connects code with a particular expectation regarding behavior at an internal software interface. Advancement in research and practice could lead to chains of evidence

[16] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

that could both support quality claims and protect trade secrets or proprietary technology in components. For example, traceability links can include modeling with respect to various attributes, as well as analyses that link models with each other and with code. These links, in aggregate, can create chains of evidence as noted above.

Software assurance (and producibility generally) are influenced not only by the *extent* of this design-related information but also by the means by which it is represented. There are four dimensions of representation that are most significant—formality (precise structure and meaning), modeling (reasoning about diverse aspects of a system), consistency (among various artifacts), and usability (feasibility for use by working development teams).

Finding 4-1: The feasibility of achieving high assurance for a particular system is strongly influenced by early engineering choices, particularly architectural and tooling choices.

Finding 4-2: Assurance is facilitated by advances in diverse aspects of software engineering practice and technology, including modeling, analysis, tools and environments, traceability, programming languages, and process support. Advances focused on simultaneous creation of assurance-related evidence with ongoing development effort have high potential to improve the overall assurance of systems.

Because modern systems of all kinds draw on diverse components from diverse sources, there will necessarily be differences in the levels of trust conferred on both components and suppliers. This means that, in the parlance of cybersecurity, there are potential attack surfaces from within the software application as well as from the outside and that we must support rigorous defense at the interfaces within the application. In other words, the new perimeter is within the application rather than around it or its platform.

Recommendation 4-1: Effective incentives for preventive software assurance practices and production of evidence across the lifecycle should be instituted for prime contractors and throughout the supply chain.

Recommendation 4-2: The DoD should expand its research focus on and its investment in both fundamental and incremental advances in assurance-related software engineering technologies and practices.

Recommendation 4-3: The DoD should examine commercial best practices for more rapidly transitioning assurance-related best practices into development projects, including contracted custom development, supply-chain practice, and in-house development practice.

5. REINVIGORATE DOD SOFTWARE ENGINEERING RESEARCH

The committee identified seven technology areas where research progress would make a difference for DoD's software capability.

- *Architecture modeling and architectural analysis.* Goals: (1) Facilitation of early validation for architecture decisions, including measures, modeling and evaluation, and compliance. (2) Facilitation of architecture-aware systems management, including models of congruence and a means to manage rich supply chains, ecosystems, and infrastructure. (3) Facilitation of component-based development, including architectural designs for particular domains.
- *Assurance: validation, verification, and analysis of design and code.* Goals: (1) Effective evaluation for critical quality attributes. (2) Assurance for components in large heterogeneous systems. (3) Enhanced

portfolio of preventive methods to achieve assurance, ranging from process improvement and architectural building blocks to programming languages and coding practice.

- *Process support and economic models for assurance and adaptability.* Goals: (1) Enhanced process support for assured software development. (2) Models for evidence production in software supply chains. (3) Application of economic principles to process decision making.
- *Requirements.* Goals: (1) More expressive models and supporting tools for both functional and quality attributes. (2) Improved support for traceability and early validation.
- *Language, modeling, coding, and tools.* Goals: (1) Enhanced expressiveness of programming languages to address current and emerging challenges. (2) Enhanced ability to exploit modern concurrency, including shared memory multicore and scalable distributed memory. (3) Enhanced developer productivity for new development and evolution.
- *Cyber-physical systems.* Goals: (1) Accelerated development of new conventional architectures for control systems. (2) Improved architectures for a wide range of embedded applications.
- *Human-system integration.* Goal: (1) Development of engineering practices for complex systems in which humans play critical roles. This area is elaborated in another NRC report.[17]

The committee made its selection of these seven technical areas on the basis of four considerations: (1) capabilities identified to have significant potential value through the committee's examination of the key DoD software producibility priorities: process, measurement, architecture, and assurance, as reported in Chapters 2, 3, and 4; (2) capabilities that can be feasibly developed through a well-managed research program, based on accepted research management criteria (such as the Heilmeier questions for research program managers who propose new program ideas—see Chapter 5); (3) capabilities not addressed sufficiently by other federal agencies; and (4) capabilities that might not develop at a sufficient pace without explicit added investment. The proposed research would be undertaken by a mix of academia, government labs, and industry. Academic research has historically had a particular role in advancing DoD technical capability, through both research and expertise, and this role persists for software producibility.

Finding 5-1: Academic research and development continues to be the principal means for developing the most highly skilled members of the software workforce, including those who will train the next generation of leaders, and for stimulating the entrepreneurial activity that leads to disruptive innovation in the information technology industry. Both academic and industry labs are creating the fundamental advances in knowledge that are needed to drive innovation leadership in new technologies and to advance software technologies that are broadly applicable across industry and the DoD supply chain.

Directions and priorities for university-originated invention are greatly influenced by funding levels and agency priorities. For example, the Defense Advanced Research Projects Agency's (DARPA's) deliberately strong relationship with the information technology (IT) research community, which began in the 1960s and endured for nearly 40 years, profoundly influenced IT research priorities, the overall culture of computer science research, and the substantial economic and social outcomes that resulted. This relationship is documented in NRC reports that trace the origins of IT innovations, each of which has led to a multibillion-dollar market.[18]

[17] See NRC, Richard Pew and Anne Mavor, eds., 2007, *Human-System Integration in the System Development Process: A New Look*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/openbook.php?record_id=11893. Last accessed August 20, 2010.

[18] See NRC, 2003, *Innovation in Information Technology*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=10795. Last accessed August 20, 2010. See also the predecessor report, NRC, 1995, *Evolving the High Performance Computing and Communications Initiative*, Washington, DC: National Academy Press. Available online at http://www.nap.edu/catalog.php?record_id=4948. Last accessed August 20, 2010.

Data from the Networking and Information Technology Research Development (NITRD) program and other sources indicate that there has been a significant reduction in federally sponsored research related to software producibility. From 2004 to 2010, overall funding for the NITRD program more than doubled. During the same period, the combined dollar allocation to the two categories most relevant to software producibility was reduced by almost half.[19] (See Box 1.5 for details.) Expressed as a percentage of the total NITRD budget, the combined allocation for the software-related categories dropped from 24.6 percent to 6.5 percent. Furthermore, it is the committee's impression that in recent years, as a consequence of these reductions, many of the researchers in these areas have moved into other fields or scaled down their research efforts.

Recommendation 5-1: The DoD should take immediate action to reinvigorate its investment in software producibility research. This investment should be undertaken through a diverse set of research programs throughout the DoD and should include academia, industry labs, and collaborations.

It is important that researchers understand the challenges associated with the way the DoD develops software.[20] This includes not only the particular technical challenges, but also the influences of factors such as the arm's-length relationship between the DoD and the contractors doing the development. DoD research agencies have instituted programs to help younger faculty get the needed domain exposure. These are important to continue and broaden if university programs are to be relevant.

Finding 5-2: Technology has a significant role in enabling modern incremental and iterative software development practices at levels of scale ranging from small teams to large distributed development organizations.

There are significant and particular difficulties in managing research in topics related to software producibility. But there are also major opportunities based on recent progress in the field, including technology developments, scientific practice, and the overall environment of production practice. Taking challenges and opportunities together, the influences include (1) the maturation of software engineering as a discipline, leading to improved research methods and lower risk in technology transition—facilitating more satisfactory responses to the Heilmeier questions; (2) the complexity of diffusion pathways and the variability of timescales, where some results can readily transfer to DoD practice, while others, often the most significant and influential, take longer and have more indirect pathways; (3) an emerging concept of novelty that is often more closely tied with readiness with respect to infrastructure and the various exponential curves than with specific technical novelty—the question is often, What are the ideas whose time has come? (4) improved methods to assess progress in the absence of crisp quantitative measures of performance (e.g., how to assess the benefits of strong typing in a quantitative way) or when the focus of research is on developing such measures; and (5) unpredictability in the span of time from the emergence of a new idea to the readiness to transition that idea with respect to practice, infrastructure, and other variables.

An additional difficulty is the development of models of return on investment in research related to software producibility. This difficulty is present for all investment in basic science and exploratory development, but it can be particularly vexing for computing technology and software. This difficulty has been the subject of intense study by the National Research Council and other groups; several reports have been produced that offer a historical perspective, showing the emergence of multiple multibillion-dollar

[19] These categories are Software Design and Productivity (SDP) and High Confidence Software and Systems (HCSS). The reported amounts for SDP and HCSS do not include 2010 NIH funding for accounting reasons that are explained in Chapter 1. Comparisons are in constant dollars.

[20] A brief description of such challenges can be found on p. 23 in NRC, 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 20, 2010.

markets on the basis of initial investment in worthy projects by NITRD agencies. Under the auspices of professional societies, similar studies were published relating specifically to software engineering research.[21] These reinforce the extent of the impact of well-managed investments.

Recommendation 5-2: The DOD should take action to undertake DoD-sponsored research programs in the following areas identified as critical to the advancement of defense software producibility: (1) architecture modeling and architectural analysis; (2) assurance: validation, verification, analysis of design and code; (3) process support and economic models for assurance and adaptability; (4) requirements; (5) language, modeling, coding, and tools; (6) cyber-physical systems; and (7) human-systems integration.

[21] Mary Shaw, 2002, "The Tyranny of Transistors: What Counts about Software?" *Proceedings of the Fourth Workshop on Economics-Driven Software Engineering Research*, IEEE Computer Society, pp. 49-51; Barry Boehm, 2006, "A View of 20th and 21st Century Software Engineering," *Proceedings of the 28th International Conference on Software Engineering*, ACM, pp. 12-29; and Leon J. Osterweil, Carlo Ghezzi, Jeff Kramer, and Alexander L. Wolf, 2008, "Determining the Impact of Software Engineering Research on Practice," *IEEE Computer* 41(3):39-49.

BOX S.2
Compilation of Report Findings and Recommendations

Chapter 1

Finding 1-1: Software has become essential to a vast range of military system capabilities and operations, and its role is continuing to deepen and broaden, including interlinking diverse system elements. This creates both benefits and risks.

Finding 1-2: The growth in the role of software in systems is due to a combination of technological advances and a maturing of the supply chain structure associated with software systems development at all levels of scale.

Finding 1-3: The DoD relies fundamentally on mainstream commercial components, supply chains, and software ecosystems for both business systems and many mission systems. Nonetheless, the DoD has special needs in its mission systems driven by the growing role of software in systems. As a result, the DoD needs to address directly the challenge of building on and, where appropriate, contributing to the development of mainstream software that can contribute to its mission.

Finding 1-4: The DoD's needs will not be sufficiently met through a combination of demand-pull from the military and technology-push from the defense or commercial information technology sectors. The DoD cannot rely on industry alone to address the long-term software challenges particular to defense.

Recommendation 1-1: The DoD, through its Director of Research and Engineering (DDR&E), should regularly undertake an identification of areas of technological need related to software producibility where the DoD has "leading demand" and where accelerated progress is needed to support the defense mission.

Finding 1-5: It is dangerous to conclude that we are reaching a plateau in capability and technology for software producibility. To avoid loss of leadership, the DoD will need to become more fully engaged in the innovative processes related to software producibility.

Chapter 2

Finding 2-1: Modern practice for innovative software systems at all levels of scale is geared toward incremental identification and mitigation of engineering uncertainties, including requirements uncertainties. For defense software, the challenge is doing so at a larger scale and in ways that are closely linked with an overall systems engineering process.

Finding 2-2: The prescription in DoD Instruction 5000.02 for the use of evolutionary development needs to be supplemented by the development of related guidance on the use of such practices as time-certain development, requirements prioritization, evidence-based milestones, and risk management.

Finding 2-3: Extensions to earned value management models to include evidence of feasibility and to accommodate practices such as time-certain development are necessary conditions to enable successful application of incremental development practices for innovative systems.

Finding 2-4: Research related to process, measurement, architecture, and assurance can contribute to the improvement of measurement practice in support of both routine management of engineering risks and value assessment as part of earned value management.

continued

BOX S.2 Continued

Recommendation 2-1: The DoD should take aggressive actions to identify and remove barriers to the broader adoption of incremental development methods, including iterative approaches, staged acquisition, evidence-based systems and software engineering, and related methods that involve explicit acknowledgment and mitigation of engineering risk.

Recommendation 2-2: The DoD should take steps to accumulate high-quality data regarding project management experience and technology choices that can be used to inform cost estimation models, particularly as they apply to innovative software development projects.

Finding 2-5: Architectural expertise is becoming dramatically more important for the DoD, its advisors, and its contractors. There will be significant and immediate benefits from advances in the state of technical support for architecture.

Recommendation 2-3: Update procurement, contracting, and governance methods to include an early and explicit architecture phase that reduces the predominant uncertainties in software intensive systems.

Recommendation 2-4: Define architectural leadership roles for major SIDRE projects and provide program managers with channels for architectural expertise.

Recommendation 2-5: Develop the technical and management infrastructure necessary to simultaneously support stabilized, high-assurance development of the current evolutionary increment while concurrently evolving the plans and specifications for stabilized development of the next high-assurance increment.

Finding 2-6: The DoD has a growing need for software expertise, and it is not able to meet this need through intrinsic resources. Nor is it able to fully outsource this requirement to DoD primes. The DoD needs to be a smart software customer. This need is particularly significant for large-scale innovative software-intensive projects for which there are cross-cutting software architectural requirements and validation challenges.

Chapter 3

Finding 3-1: Industry leaders attend to software architecture as a first-order decision, and many follow a product-line strategy based on commitment to the most essential common software architectural elements and ecosystem structures.

Finding 3-2: The technology for definition and management of software architecture is sufficiently mature, with widespread adoption in industry. These approaches are ready for adoption by the DoD, assuming that a framework of incentives can be created in acquisition and development efforts.

Finding 3-3: The DoD would benefit from explicit attention to software architecture and industry best practice, including (1) formalizing career paths and role descriptions for software architects, (2) identifying ways that DoD-aligned software architects can provide objective advice (see Chapter 2), and (3) enhancing organizational structures to support effective architectural leadership.

Finding 3-4: Several DoD programs are using software architecture-driven acquisition with successful results.

BOX S.2 Continued

Recommendation 3-1: Initiate a targeted research program to provide software architects with better tools and techniques for DoD systems.

Recommendation 3-2: This committee reiterates the past Defense Science Board recommendations that the DoD follow an architecture driven acquisition strategy, and, where appropriate, use the software architecture as the basis for a product-line approach and for larger-scale systems potentially involving multiple lead contractors.

Recommendation 3-3: The DoD should enhance existing practices to afford better distinctions between precedented portions of systems and innovative portions of systems, wherein architectures are developed both to encapsulate the innovative elements and to afford maximum opportunity to build on experience and existing ecosystems for precedented elements. These overall architectures, and particularly the innovative elements, should be subject to early and continuous validation, especially in systems that have requirements for interoperation.

Finding 3-5: In systems with innovative functional or quality requirements, benefit is derived from an early focus on the most essential architectural commitments and quality attributes, with deferred commitment to specifics of functional characteristics. This approach can reduce the overall uncertainty of the engineering process and yield better outcomes.

Recommendation 3-4: The DoD should learn from commercial experience and, in addition, sponsor diverse areas of technical research to help reduce the engineering risk in architecting systems that include unprecedented functional and quality attributes.

Chapter 4

Finding 4-1: The feasibility of achieving high assurance for a particular system is strongly influenced by early engineering choices, particularly architectural and tooling choices.

Finding 4-2: Assurance is facilitated by advances in diverse aspects of software engineering practice and technology, including modeling, analysis, tools and environments, traceability, programming languages, and process support. Advances focused on simultaneous creation of assurance-related evidence with ongoing development effort have high potential to improve the overall assurance of systems.

Recommendation 4-1: Effective incentives for preventive software assurance practices and production of evidence across the lifecycle should be instituted for prime contractors and throughout the supply chain.

Recommendation 4-2: The DoD should expand its research focus on and investment in both fundamental and incremental advances in assurance-related software engineering technologies and practices.

Recommendation 4-3: The DoD should examine commercial best practices for more rapidly transitioning assurance-related best practices into development projects, including contracted custom development, supply chain practice, and in-house development practice.

continued

BOX S.2 Continued

Chapter 5

Finding 5-1: Academic research and development continues to be the principal means for developing the most highly skilled members of the software workforce, including those who will train the next generation of leaders, and for stimulating the entrepreneurial activity that leads to disruptive innovation in the information technology industry. Both academic and industry labs are creating the fundamental advances in knowledge that are needed to drive innovation leadership in new technologies and to advance software technologies that are broadly applicable across industry and the DoD supply chain.

Finding 5-2: Technology has a significant role in enabling modern incremental and iterative software development practices at levels of scale ranging from small teams to large distributed development organizations.

Recommendation 5-1: The DoD should take immediate action to reinvigorate its investment in software producibility research. This investment should be undertaken through a diverse set of programs across the DoD and should include academia, industry labs, and collaborations.

Recommendation 5-2: The DoD should take action to undertake DoD-sponsored research programs in the following areas identified as critical to the advancement of defense software producibility: (1) architecture modeling and architectural analysis; (2) assurance: validation, verification, analysis of design and code; (3) process support and economic models for assurance and adaptability; (4) requirements; (5) language, modeling, coding, and tools; (6) cyber-physical systems; and (7) human-systems integration.

1

Recognize the Pivotal Role of DoD Software Innovation

THE ROLE OF SOFTWARE IN DEFENSE

The pivotal role of information technology (IT) in defense has been noted in multiple studies.[1,2,3,4] Software is increasingly used to embody the functionality of defense systems of all kinds,[5] and IT is used pervasively in the Department of Defense (DoD) for a multitude of different purposes and in a multitude of different program types (Box 1.1). It is a key enabler of overall systems scale and complexity, of integration among systems (net-centricity and "ultra-scale"), and of agility in systems. Mission capability embodied in software has become a unique source of strategic and military advantage, and software producibility is emerging as a key component of military strength, capability, and readiness. The committee uses the term "software producibility" to refer to the capacity to design, produce, assure, and evolve software-intensive systems in a predictable manner while effectively managing risk, cost, schedule, and complexity.

The Defense Science Board's (DSB's) Task Force on Mission Impact of Foreign Influence on DoD Software, which explored the essential role of software in defense, released its report in September

[1] Defense Science Board (DSB), September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[2] DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA385923. Last accessed August 20, 2010.

[3] National Research Council (NRC), 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, DC: National Academies Press.

[4] NRC, 1999, *Realizing the Potential of C4I: Fundamental Challenges*, Washington, DC: National Academy Press. Available online at http://www.nap.edu/catalog.php?record_id=6457. Last accessed August 20, 2010.

[5] DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA385923. Last accessed August 20, 2010.

BOX 1.1
A Taxonomy of Information Technology Program Types

Many taxonomies have emerged to assist managers in identifying common patterns among requirements and system types and then building on that knowledge to define and implement best practices and standards that are suited to particular categories of systems. Distinctions are made based on the function and role of the system, the types of risk to be addressed, the scale of systems and budgets, and other factors.

This report adopts the following taxonomy of IT programs:

1. Business systems and office IT
2. Command and control
3. Computing and communications infrastructure
4. Intelligence, Surveillance, and Reconnaissance (ISR), space, and weapons

The classification, which is loosely based on a classification scheme used within the DOD to track IT acquisition programs,[1] is primarily functional. But the functional categories also correspond roughly to distinctions among programs based on the extent of innovation (more in categories 2 and 4) and those that are more likely to have precedented requirements and architectures and thus build on established ecosystems (categories 1 and 3).

These categories also separate IT that is embedded in weapons or weapons systems or similar platforms with potentially high systems risk (category 4), IT in which software and hardware are less tightly integrated (categories 1 and 2), and IT that provides the computing and communications infrastructure (category 3) that can be used by systems identified in the other categories.

Because modern larger-scale systems are interconnected and therefore more often integrate across these functionalities, greater numbers of systems may cross these boundaries. For example, many weapons systems incorporate command-and-control functionalities.

Finally, despite the differences among these categories, most systems rely on similar development practices, including design and architectural concepts, programming languages, process and measurement concepts, and tools.

[1] Based on a taxonomy used by the Office of the Assistant Secretary of Defense for Networks and Information Integration to categorize major automatic information system programs.

2007.[6] The report notes that "in the Department of Defense, the transformational effects of information technology (IT—defined here broadly to include all forms of computing and communications), joined with a culture of information sharing, called Net-Centricity, constitute a powerful force multiplier. The DoD has become increasingly dependent for mission-critical functionality upon highly interconnected, globally sourced, IT of dramatically varying quality, reliability and trustworthiness."[7] In other words, *at the core of the ability to achieve integration and maintain agility is the ability of the DoD to produce and evolve software*. This echoes a judgment expressed in many other studies that have considered the role of software in defense.[8] The report further notes, however, that "each year the Department of Defense depends more on software for its administration and for the planning and execution of its missions,"

[6] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[7] Ibid., p. vii.

[8] See referenced studies above.

and additionally "this growing dependency is a source of weakness exacerbated by the mounting size, complexity, and interconnectedness of its software programs."[9]

The rapid growth of software in defense systems is especially significant and parallels the growing role of software in a broad range of application domains. This growth is a natural outcome of the special engineering characteristics of software: Software is uniquely unbounded and flexible, having relatively few intrinsic limits on the degree to which it can be scaled in complexity and capability. Software is an abstract and purely synthetic medium that, for the most part, lacks fundamental physical limits and natural constraints. For example, unlike physical hardware, software can be delivered and upgraded electronically and remotely, greatly facilitating rapid adaptation to changes in adversary threats, mission priorities, technology, and other aspects of the operating environment. The principal constraint is the human intellectual capacity to understand systems, to build tools to manage them, and to provide assurance—all at ever-greater levels of complexity.

The extent of the DoD code in service has been increasing by more than an order of magnitude every decade, and a similar growth pattern has been exhibited within individual, long-lived military systems. In addition to this growth in size (as well as growth in other system aspects such as resource usage), there is a corresponding growth in overall system capability and complexity.

This chapter addresses the first two of the four questions taken up by the committee:

- To what extent is software capability significant for the DoD? Is it becoming more significant or less so?
- Will the advances in software producibility needed by the DoD emerge unaided from industry at a pace sufficient to meet evolving defense requirements?

Growth in the Role and Significance of Software to Defense

The value that software contributes to major systems is increasing rapidly and becoming more fundamental to system capability. The DSB Task Force report on defense software (2000)[10] illustrates this point in the case of combat aircraft. The percentage of system functions performed by software has risen from 8 percent of the F-4 in 1960, to 45 percent of the F-16 in 1982, to 80 percent of the F-22 in 2000.[11] Software has become essential to all aspects of military system capabilities and operations, and software-specific investment is critical to them.[12]

Macroeconomic data show analogous growth in the role software plays in the commercial world. This is significant because commercial vendors are key contributors to the defense software supply chain—for Future Combat Systems,[13] for example, 27 million source lines of code (more than 42 percent

[9] DSB, September 2007, *Report of the Defense Science Board Task Force on Defense Software*, p. v.

[10] DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA385923. Last accessed August 20, 2010.

[11] Ibid., Table 3.3a. Available online at http://www.acq.osd.mil/dsb/reports/defensesoftware.pdf. Accessed February 25, 2008.

[12] Boehm, Kind, and Turner quote an unidentified U.S. Air Force General, "About the only thing you can do with an F-22 without software is take a picture of it." In Barry Boehm, Richard Turner, and Peter Kind, 2002, "Risky Business: Seven Myths About Software Engineering That Impact Defense Acquisitions," *Program Manager*, May 1, 2002. The committee notes, however, that with modern cameras, even taking a picture cannot be done without software.

[13] Future Combat Systems (FCS) was "the Army's modernization program consisting of a family of manned and unmanned systems, connected by a common network, that enables the modular force, providing our Soldiers and leaders with leading-edge technologies and capabilities allowing them to dominate in complex environments." U.S. Army, "Future Combat Systems." Available online at http://www.army.mil/fcs/. Accessed March 3, 2008.

of the total delivered executable source lines of code) were commercial off-the-shelf (COTS) or open source.[14]

It is also significant because software capability has become a strategic source of market differentiation in many industries, from financial services and health care to telecommunications and entertainment. A 2002 report by the National Research Council's (NRC's) Board on Science, Technology, and Economic Policy[15] noted that since 1995 the IT and networking industries had accounted for 20 percent of the nation's economic growth, even though they accounted for only 3 percent of gross domestic product (GDP). Comparable figures exist in the European Community—the information and communications technology (ICT) sector represents just above 5 percent of the European GDP, but reports show that ICT drives 25 percent of overall growth and about 40 percent of the increase in productivity.

Finding 1-1: Software has become essential to a vast range of military system capabilities and operations, and its role is continuing to deepen and broaden, including interlinking diverse system elements. This creates both benefits and risks.

Software in Systems

Military system capability is heavily dependent on software, which has become an enabler for much of the functionality and flexibility of our war-fighting systems. Software has proven to be a differentiator in system capability for a wide range of current systems such as the F-22, F-35 Lightning II, and the Aegis Combat System. Software to modify and integrate existing capabilities was a key enabler in the February 2008 successful shoot-down of an errant U.S. satellite as it tumbled back to Earth.

This critical role of software in defense is also noted in the more recent DSB Task Force report on foreign software, which states, "The DoD now relies upon networked, highly-interconnected systems for many mission-critical capabilities, and this reliance is projected to increase. The software in these systems is the key ingredient that provides much of the increased capability delivered to the warfighter, just as it represents the key factor in increased productivity and new capabilities for industry today. For the DoD, this advanced technology is a force multiplier."[16,17] A high level of software capability is also important in producing defense systems. For example, very-large-scale, highly networked, and crypto-secured software systems were needed for the robotic design used to construct the production line for F-35 manufacturing.

Given the importance of software to the DoD, a vital question is how the department can ensure that it will be able to meet its software needs now and into the future. The subsequent chapters of this report explore significant facets of this issue.

Software provides the means to manifest the modeling and simulation capability now essential in the design and testing of advanced military platforms and weapons in all branches of the DoD. These design-focused software capabilities can save millions of dollars—all before the first piece of metal is bent. But perhaps even more importantly, in these early stages this software capability enables the customer to focus on driving out risks related to the definition of weapons and systems functionality and

[14] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, p. 77. Available at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 10, 2010. The FCS program was cancelled in 2009, but the experience of that program nonetheless provides valuable insight.

[15] NRC, 2002, *Measuring and Sustaining the New Economy: Report of a Workshop*, Washington, DC: National Academies Press, p. 52. Available online at http://www.nap.edu/openbook.php?record_id=10282&page=52. Last accessed August 10, 2010.

[16] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, p. 12. Available at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[17] DSB, April 2009, *Creating a DoD Strategic Acquisition Platform*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA499566&Location=U2&doc=GetTRDoc.pdf. Last accessed August 20, 2010.

architecture. The savings are due to the elimination of the need to build and test numerous prototype designs—this is now done through software. For many systems, this modeling and simulation software implements high-fidelity, massively parallel computational fluid dynamics (CFD) simulations. These simulations enable experimentation with signatures of different missile and aircraft designs without having to resort to expensive physical tests. Wind tunnel testing alone, for example, can cost millions of dollars per week, with months of testing required to settle design issues.

Other examples of software-manifest functionality used pervasively in defense systems include onboard prognostic health systems and automated logistic support systems, both of which emerged from Defense Advanced Research Projects Agency (DARPA) research funding in the 1980s. These software-enabled systems are quietly saving millions of dollars.

Software in Organizations

In addition to deepening the extent of reliance on software in systems components and in the tooling associated with their development, there is a significant broadening of the role of software in systems and organizations. These changes reflect both a growing centrality in the role of software and also a growing portfolio of associated risks. This combination raises the significance of software-related decisions in systems development—software has become the medium of choice for innovative functionalities. These functionalities include not only component capabilities, but also functionalities that are enabled through the use of software to implement interconnections across a family of constituent systems. However, largely as a consequence of both of these roles—innovation and interconnection—software has emerged as the locus for a range of engineering challenges related to reliability, security, and development predictability.

The interconnection aspect of this shift has three principal elements. First, there is a shift in emphasis from the development of functionally focused systems to the development of systems that interconnect and integrate capabilities within and across enterprises. The "net-centric" and "ultra-scale" concepts are reflective of this shift. This has great strategic benefit, in that it leverages the value of dispersed assets and enables agile responses at a broad range of echelons in war-fighting situations when multiple systems elements are involved. This interconnection has associated risks, of course, primarily related to the magnitude of failures experienced as a consequence of internal errors, vulnerabilities exploited, etc. In civilian systems, for example, there are widely reported examples of cascading failures of interconnected systems in telecommunications, utilities, and supply chain systems.

Second, largely as a consequence, IT staffs are generally less involved in mediating between a system and its users. Much larger numbers of DoD personnel interact directly with systems, and indeed many systems can be (and need to be) accessed through public communication infrastructure. This is more efficient because it removes the delays and inaccuracies caused by intermediation. But it also means that many more individuals—usually inadvertently, but not always—can take actions with wide-reaching consequences, both positive and negative.

A third element of the interconnection aspect is that modern systems can support immediate electronic enactment of decisions. This enables agility and fast response in decisions and actions—getting inside the command loop of an adversary, but it also means that failures and compromises can happen very quickly, inside a human decision loop. An example in the civilian context is the recent discussion over the duration in milliseconds of the stock trading look-ahead window.

Fourth, interconnection introduces new information security challenges.

Software Supply Chains

The growth in the role of software, as described above, is enabled in part by a surprisingly recent phenomenon in IT, which is the diversification and enrichment of the supply-chain structure for IT systems. This enrichment is more than just systems outsourcing as experienced in the past half century.

Supply chains for software systems are both broader and deeper today, and they include commercial as well as defense players and involve technically rich and complex architectures, with frameworks, libraries, services, and other roles contributed by multiple players. The value of outsourcing, which was initially primarily cost reduction and access to expertise, now includes greater agility and ability to respond to changes in the operating environment.

The supply-chain structure for modern defense software is evolving in a similar way, and is now significantly more complex and more international than it was even just a decade ago. Indeed, this combination of factors motivated the Defense Science Board study mentioned above to assess the impact of this internationalization on defense software systems, including their development and their assurance.[18] The complexity—and the internationalization—are due to a combination of factors, including certain technical developments in software technology, the economic forces and technological enablers of globalization, the geographic dispersion of the trained workforce, the minimal capital investment required (not including education and training) for the workforce, and increasing demand for precedented (routinized) projects. The complexity is also enabled by the maturation and acceptance of a diverse set of commercial (COTS) ecosystems with their associated components and infrastructure.

From a technological perspective, this richness in the supply chain is enabled by advances in both organizational collaboration technologies and software technology. The collaboration technologies build on Internet infrastructure to provide, for example, messaging, process support, team information servers (document sharing and configuration management), issue databases, servers for software builds, wikis, automated test and analysis tools, and the like.

Finding 1-2: The growth in the role of software in systems is due to a combination of technological advances and a maturing of the supply chain structure associated with software systems development at all levels of scale.

PRECEDENT AND INNOVATION IN SOFTWARE

Precedented Software and Externalities

Software development today relies heavily on established architecture and infrastructure component configurations, which the committee calls *software ecosystems* (see Boxes 1.2, 1.3, and 1.4 for elaboration). The success of the ecosystem model derives from the natural convergence of component functionalities and the associated architectural elements, chiefly protocols and software/service interfaces, through which these functionalities are delivered within applications and systems. These functionalities are called precedented, in the sense that new users of these functionalities benefit, in terms of design costs and risks, from the experience of existing and prior users. Once ecosystems are established, the development processes associated with them are often characterized primarily by selection of an ecosystem and then, within that ecosystem, tailoring through configuration of settings and the authoring of a relatively very small amount of custom software code. Thus, custom development in these areas of convergence gives way to product selection and procurement. Indeed, because engineering risks are relatively modest, a straight-line sequential process may often be appropriate for development management in these precedented portions of larger systems.

The emergence of generally accepted ecosystem structures has in recent years become one of the enablers of the growth in richness in the supply-chain structure for software systems, which in the commercial world has promoted a diversity of suppliers, growth of a market for specialists in component-level innovation, and tools geared to development productivity for particular ecosystems. A

[18] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

countervailing trend in recent years has been the consolidation at the top end of the software vendor market through mergers and acquisitions.

How does this ecosystem phenomenon influence the DoD? First, it should be clear that the DoD derives huge benefit from the ecosystem phenomenon, because so many of the business systems required by the DoD have requirements that are strongly analogous to commercial requirements. In these cases, only modest adaptation of commercial best practices, including choices and configuration of ecosystems, can yield major advantage to the DoD in the form of low costs, managed risk, and predictable outcomes. Additionally, the software elements of an ecosystem are complemented by a supplier ecosystem of expert integrators, consultants, add-in vendors, and the like. In other words, when the DoD's requirements are similar to commercial requirements, the DoD benefits significantly by adopting commercial best practices where possible. (See Boxes 1.2 and 1.3 for elaboration on the concept and role of ecosystems.)

Not only are engineering risks reduced for precedented developments, but also there are benefits from the richness of the supply chain due to network effects—the positive externalities associated with systems adoption.[19] When an ecosystem is successful, the commonalities of structures and interfaces enable larger numbers of organizations to participate efficiently in the development of large systems, providing software components, libraries, frameworks, plug-ins, custom elements, and so on. This makes it possible for more suppliers to participate and also reinforces the status of accepted frameworks, broadening the benefits for framework adopters and affording them the opportunity to make good choices at the component level while working within the safe conventions of an established ecosystem.

Indeed, for many of these categories of requirements, it is nearly intractable, from the standpoint of cost and risk, to develop "separate but equal" approaches. Not only are these expensive, time-consuming, and risky, but also the DoD would then need to bear the entire cost of advancing the idiosyncratic technology, whereas ecosystem participants would benefit from actions by other participants to incrementally advance the performance and capability of the ecosystem, its infrastructure, and its constituent components. (Of course there are also negative externalities associated with widely adopted ecosystems, such as their attractiveness to developers of security exploits and the consequent ease of access by adversaries to offensive capability.)

Innovative Requirements and the Activity of Innovative Software Development

The situation becomes more complex and challenging when the DoD requires functionality more specifically focused on the defense mission—weapon systems, command and control systems, intelligence analysis systems, and other systems more directly supportive of war-fighting and intelligence. They include high-performance embedded systems, large-scale systems with unprecedented architecture, and highly interconnected systems. They often require high degrees of software assurance. The functionalities of these systems are more specialized and also, due to the presence of ambitious adversaries, constantly evolving in response to changing threats. They require the management of complex and evolving requirements.

Many of these systems are less precedented in the sense that innovation is required in system architecture, design, infrastructure, linkages with hardware sensors and effectors, and other respects. Technological enablers for the development of these more innovative systems—either with respect to innovative functionality or innovative engineering or both—is the principal focus of this report. This does not exclude considerations regarding precedented systems, however, because larger systems almost always involve a mix of innovative and precedented functionalities and components. Indeed, there are few modern defense systems of scale that do not build on technologies extensively drawn from related defense systems and from the various mainstream software ecosystems. This means that development and acquisition practices must account for this mix of the innovative and the precedented. This mix

[19] Carl Shapiro and Hal R. Varian, 1998, "Information Rules: A Strategic Guide to the Network Economy," *Journal of Technology Transfer* 25(2):250-253.

BOX 1.2
The Concept of Software Ecosystems

In web applications, there are conventional configurations of server operating systems, relational databases, web servers, application server frameworks, business rules, and other elements that are combined to create e-commerce servers. These servers rely, in turn, on the "rich client" ecosystem of a modern client-side web browser, which includes not just HTML and basic HTTP, but also technologies such as JavaScript, XML, DOM access, and asynchronous HTTP.

Analogous configurations, with very different sets of interfaces and components, are used to support mobile applications (Apple's iPhone ecosystem and Google's Android ecosystem are two recent examples), business intelligence (OLAP, etc.), enterprise resource planning, and other common functionalities. There are competing ecosystems in the commercial world—for example, web application servers can be developed using Java-based ecosystems such as the platform-independent Java EE (formerly known as J2EE) or using the Windows-based .NET Framework, which supports multiple programming languages sharing common runtime services related to memory management, security, etc. In web applications, there are also open source ecosystems, one of which is the so-called LAMP stack, which comprises the Linux operating system, the widely adopted Apache web server, the MySQL relational database, and scripting in a language starting with the letter "P," most usually PHP, Python, or Perl. The open-source character of this ecosystem means effectively that it operates as a kind of quasi-consortium linking the various stakeholders that participate in the ongoing development of the overall architecture, the details of the interfaces, and the code base. Regardless of particulars, in most of these ecosystems, choice of programming language is often driven by the choice of ecosystem.

The committee defines a software ecosystem as a conventional structure consisting of a family of infrastructural elements that are intended to be combined in a patterned way. Ecosystems include software-architectural structure, but they can also include configurations of hardware and services platforms. Ecosystems generally provide a reuse of major elements and infrastructure, which can entail strong structural and semantic commitments.[1] Ecosystems often also include documents, tools, practices, and even organizations to accompany these elements. The principal benefits include potentially significant reductions in cost, mitigation of engineering risk, and up-front agreement on representations and meanings for data that are shared within a system or across systems.

It is also significant to note that, if we broadly construct the idea of a software ecosystem, then the Internet family of protocols would also be an example. The ecosystem comprising these protocols and its evolution have been much studied—one of the results of this is the "hourglass" model. This model illustrates how there can be a diversity of means for provisioning the service associated with a particular interface or protocol, such as TCP, and a separate diversity of client applications that build on the service. For example, "TCP service" is generally provided as a "layer" above IP, which in turn can be provisioned over fiber, wireless, copper, and many other means. TCP, in turn, underlies the web protocol HTTP as well as file transfer FTP and many other higher-level services.[2]

---

[1] Not all ecosystems involve direct reuse of components. A family of protocols, for example, defines a "means of exchange" among system components. Other examples include instruction set architectures (with multiple vendors providing chips) and agreed-upon XML or other data exchange representations for shared information.

[2] See, e.g., NRC, 2001, *The Internet's Coming of Age,* Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=9823. Last accessed August 20, 2010.

Another example is the relatively small set of widely adopted ecosystem architectures for embedded and real-time control applications (such as the QNX, RTLnix, VxWorks, and Windows CE real-time operating systems), the automotive industry's AUTOSAR (AUTomotive Open System Architecture), and the SCADA protocols and interfaces used in the electrical grid. Many government users of embedded and real-time capability have taken actions to work with researchers and vendors to develop more capable ecosystems that build on modern concepts and abstractions related to processors, languages, and tools.[3,4] Ecosystems are also being adopted or are emerging in areas ranging from robotic systems to data-intensive supercomputing.[5]

The ecosystem phenomenon is now pervasively apparent in commercial industry, and it is actively promoted by leading vendors, in part due to the stability of market structure derived from the network effects. Ecosystem or framework "owners" (from the examples above: Apple, Microsoft, Google, Oracle, several open-source foundations, and many others) control the trajectory of the overall market for components and services associated with that ecosystem, but many firms participate in that "internal" market. Although the risks and costs associated with introducing a new ecosystem or framework may be very high, the risks and costs for niche providers within an established framework can be low. Additionally, once a community of suppliers is engaged within an ecosystem, the overall ecosystem can continue to evolve in response to the broad market trajectory and also to new technology developments. Thus, new languages can be added to .NET (e.g., functional programming with F#), new libraries and language features can be added within Java EE (closures to Java), and so on. This is one of the enablers and sustainers of the global supply chain.

The committee notes, in addition, that the structure of ecosystems may become more or less exposed to developers and users, and indeed entire ecosystems may split or merge. For example, a vendor could choose to expose a previously inaccessible internal interface to allow greater customization by customers and integrators. In the case of enterprise resource planning (ERP) systems, for example, vendors expose interfaces that allow "add-in" developers to provide a diverse set of functionalities tailored to particular market segments. This enables a broader diversity of client requirements to be met with less risk to both clients and the framework vendor. The ecosystem structure thus evolves according to changing opportunities and risks for the various stakeholders. Examples of the considerations include network effects (benefits of broader adoption of particular "hourglass necks"), vertical integration (reduction in risks associated with integrating separately developed components), and lock-in (greater friction in interoperation and choices available for component functionalities).

---

[3] See, for example, the workshop convened by the NITRD High Confidence Software and Systems Coordinating Group, "National Workshop on High-Confidence Automotive Cyber-Physical Systems," April 3-4, 2008, Troy, Michigan. Available online at http://varma.ece.cmu.edu/Auto-CPS/. Last accessed August 20, 2010.

[4] See also, Jeanette Wing, 2008, "Cyber-Physical Systems Research Charge," presented at the Cyber-Physical Systems Summit, April 24, 2008, St. Louis, MO. Available online at http://www.cra.org/ccc/docs/cps-summit.pdf. Last accessed August 20, 2010.

[5] NRC, 2009, *Assessing the Impacts of Changes in the Information Technology R&D Ecosystem: Retaining Leadership in an Increasingly Global Environment*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12174. Last accessed August 20, 2010.

BOX 1.3
The Role of the Software Ecosystems for the DoD

There are a number of key advances in software technology that enable the emergence of ecosystems and other architecture-based enrichments in supply-chain structure, with consequent benefits to cost and agility and also the consequent risks of more diverse sourcing. These advances range from the design of software architectures, frameworks, and components to the technical properties of modern programming languages, including first-class encapsulation, advanced typing, interface and package structures, and framework architectures.

These and other software technology improvements have also enabled the development and successful implementation of a number of conventional structures of software components used for commercial and government applications. These conventional structures, here called ecosystems, are families of infrastructural elements that are combined in a patterned way to construct precedented applications such as many business and back-office systems, e-commerce and web applications, and mobile applications. Ecosystems include such software-architectural structure as stacks, hardware and software platforms, and software frameworks. They often also include the documents, tools, and practices that accompany these elements (e.g., Eclipse and ASP.NET). The ecosystems also include a diverse array of software and organizational services that support conventional architectural structures.

The ability to successfully define robust and broadly adoptable standards (de facto or ratified) and to stimulate a critical mass of compliant implementations is a significant enabler of ecosystem success. The economics of network externalities[1] play a significant role in reinforcing successful ecosystems (often regardless of technical merit) and in guiding the initial stages of promotion of emerging new ecosystems. (This issue is elaborated below.)

A particular challenge for the DoD in defining its own ecosystems is to keep up with rapidly evolving technology and to select those interfaces and component capabilities for which certified components and/or trusted suppliers exist. Ecosystems in different categories may share elements and typically support diverse ranges of applications. Over the years, the DoD has attempted to codify its preferences regarding the components and interfaces in these conventional aggregates with efforts such as the Defense Information Infrastructure Common Operating Environment (DIICOE), the Joint Technical Architecture (JTA), and the System of Systems Common Operating Environment (SOSCOE). Generally speaking, these are broad suites of conventionalized standards and infrastructural elements that have been judged acceptable for adoption in systems generally. These suites may identify particular acceptable ecosystems, but they are not themselves ecosystems by the committee's definition (or architectures, in the sense of Chapter 3).

[1] See, e.g., Carl Shapiro and Hal R. Varian, 1998, *Information Rules: A Strategic Guide to the Network Economy*, Boston: Harvard Business Press.

often creates confusion regarding development processes and other systems engineering choices. "One size fits all" models—even for incremental iterative developments—can be dangerous.[20]

Because of the extent to which modern software builds on existing ecosystem and infrastructural elements, modern software development processes entail activities that go well beyond the design and authoring of new code. Modern software development is much more about identifying and defining appropriate and scalable architectures; selecting, using, and adapting infrastructure such as frameworks, components, and libraries; and deploying best practices and tools for collaboration, process support, and

[20] With respect to process (Chapter 2), hybrid approaches may best be employed, for example, combining straight-line processes for precedented elements with iteration and prototyping for innovative elements. With respect to architecture (Chapter 3), modular designs that "concentrate" particular innovative or rapidly evolving functionalities in individual components can greatly reduce overall project risk.

Modern e-commerce frameworks, for example, are ubiquitous in Internet-based commerce, but they have also been adopted as internal coordination frameworks for large-scale DoD systems.

The ecosystems contribute enormous value to software and systems projects that rely on them by allowing developers to leverage an enormous investment for which costs are spread across a wide base of users rather than taking on the full effort and expense of developing an entire software system from the ground up. When ecosystems are widely adopted, architectural risks are drastically reduced because principal architectural commitments are embodied in the successful ecosystem, and, additionally, the extent of value to the DoD can grow over time due to the positive network externalities. Thus, the DoD derives benefits from ecosystem adoption, but it must be attentive both to selection criteria and to its ability to participate in the overall evolution and development of the ecosystems within which it participates. One particular issue is the sustainability of ecosystems that are adopted into systems. In some instances, choices may depend more on appraisals of sustainability and network effects (to use economic terms) than on particular technical characteristics. These sustainability factors may influence engineering risk (see Box 2.2). The DoD may derive great benefits from investing in the evolution of the ecosystems in which it participates, which enhances both technical fit and sustainability.

As noted above, these ecosystems are enabled by a wide range of computer science and software engineering advances. The modern software application frameworks essential to Web-based systems, e-commerce, and graphical user interfaces of all kinds are enabled by the same advances in programming language design that led to languages such as Java, C#, and Ada95. Many of these "component" advances and, perhaps more importantly, the principal abstractions and architectural concepts underlying established ecosystems, are legacies of past DoD investment in computing technology R&D, primarily in the form of 6.1 and 6.2 extramural research funding.

Because of the rapid pace of infrastructural development, the competitive business environment, and the need to accommodate new functionality, the ecosystems are generally in a state of continuous evolution, carefully managed to stage out new increments of value while minimizing costs and risks for existing adopters—and thus to retain the benefits of the positive externalities. The evolutionary trajectory for some ecosystems is entirely driven by particular vendors, as is the case with Microsoft and .NET or Oracle and its E-Business Suite. Others are driven by complex community processes, as in the case of the open-source LAMP stack (see above), the Internet protocols themselves, and also some commercial ecosystems, as is the case with many of the ecosystems surrounding Java—following the Java Community Process.[2] The evolution may include specific component capabilities, architectural and interfaces structures, and associated tooling (as in the case of Visual Studio and .NET).

[2] For more information, see the Java Community Process at http://jcp.org/en/home/index. Last accessed August 20, 2010.

validation. Indeed, it is generally recognized that, for large systems in industry and aerospace, the most significant costs are generally associated with gathering functional and non-functional requirements, developing architecture and design, managing process, and achieving assurance—and somewhat less with the writing and evolution of code.[21] (Issues related to innovative systems for defense are addressed in two chapters of this report—Chapter 2 focuses on requirements, and Chapter 3 focuses on architecture and agility at scale.)

[21] See, for example, economics studies by Barry Boehm, studies of the IBM Rational Unified Process, and other work that shows that the proportion of coding in the overall process is diminishing. See, RTI, 2002, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Planning Report 02-3, RTI Project Number 7007.011. Available online at http://www.nist.gov/director/planning/upload/report02-3.pdf. Last accessed August 20, 2010.

BOX 1.4
The Concept of Alignment

As noted earlier, complex evolving software ecosystems and rich supply chains are enabled by relatively recent technological developments, although the market forces have been present for a much longer time. The benefit to government and firms alike is in how software is managed, with the breakthrough result of a better alignment of IT structures with operational structures in organizations. This affords firms greater flexibility in buy-versus-build decisions. But, more importantly, it enables many organizations to outsource common infrastructure such as databases, application servers, and software frameworks, and to in-source only those critical elements that provide unique capabilities and advantage over competitors. This amounts to an "escape" from the need to redundantly develop infrastructural capabilities.

The committee uses the term "alignment" to refer to this ability to in-source only key differentiators and organization-specific capabilities. Generally speaking, alignment is achieved incrementally. As capabilities that previously were innovative become commonplace across firms, the task of advancing them (from the standpoint of a technology user organization such as DoD back-office business functions) is shifted from internal resources to external ones (vendors and other outsource suppliers), enabling the firms to redirect their internal resources to new areas where they can differentiate themselves from their competitors. This is how, for example, the central database for many firms evolved from early network and hierarchical databases into relational transactional databases into virtualized application server capabilities wrapped around web servers and relational databases, with most of the functionality being provided by outside vendors or through open-source de facto consortia as in the LAMP stack.

The DoD also benefits from this when it can shift from expensive custom components (that it must maintain throughout an entire system lifecycle) to off-the-shelf components that are constantly being improved upon by their vendors in response to market forces. Thus, as technologies evolve and "commoditize," there is a general trend to shift function from in-source to outsource.[1] An issue for the DoD, however,

[1] But this is not always the case, as new dimensions of capability and differentiation emerge for formerly commoditized infrastructural elements, as is happening now for data centers and their architectures.

As noted both in the workshop report issued by this committee[22] and in the recent Software Engineering Institute (SEI) report on ultra-scale systems,[23] these issues may be made more challenging for modern interconnected defense systems due to overall scale and complexity, and particularly in requirements and architecture. These systems experience a great deal of architectural risk due to the often long delay until the consequences of early engineering decisions are felt and understood. Additionally, the decentralized governance models that are typical for large-scale interlinked systems (ultra-scale, net-centric, system of systems) can have both positive and negative effects on risk. Also, overly conservative choices regarding how to measure progress and earned value can lead toward local optima but away from overall systems-scale success (see Chapter 2). Finally, over-commitment to particular requirements early in the process can result in lost opportunities for radical cost savings or capability improvements downstream. These risks could potentially be mitigated through innovation in both technological and process measures.

[22] NRC, 2007, *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://books.nap.edu/catalog.php?record_id=11936. Last accessed August 20, 2010.

[23] Software Engineering Institute (SEI), 2006, *Ultra-Large-Scale Systems: The Software Challenge of the Future*, Pittsburgh, PA: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf. Last accessed August 20, 2010.

as for other entities seeking to maintain leadership in software use and development, is how to effectively track the evolution of the conventional interfaces and architectures and not fall behind. Another issue, more directly related to innovation, is how to work with the broad technology community to ensure that—where the DoD has leading demand—its requirements can be met as the infrastructure evolves. Historically, the DoD has accomplished this in many core IT areas, as noted in multiple studies. [2,3,4]

Indeed, when infrastructures are traced historically, it is evident that many of the fundamental architectural concepts originated in or were stimulated by DoD-sponsored research. For example, many of the architectural elements of Linux can be traced back to BSD Unix and even Multics. This point relates to the discussion later in the report regarding the role of the DoD in architecture.

From the defense perspective, this yields both benefit and risk. Expertise in critical component and infrastructural technologies is concentrating, and the capability of those components is rapidly advancing. The DoD cannot so easily "build" when other players are all "buying," even when there may be assurance challenges with respect to component providers. If the DoD makes too many "build" decisions for infrastructural capabilities, costs and risks escalate to the point of intractability. This is because the DoD would have to bear the entire cost and risk of developing and sustaining its own custom version of the technology. The established implementations of that technology may have been evolving in the larger commercial market over a period of years, with effective investments spread across a multitude of vendors and users.

[2] NRC, 1997, *Ada and Beyond: Software Policies for the Department of Defense*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=5463. Last accessed August 20, 2010.

[3] NRC, 2000, *Making IT Better: Expanding Information Technology Research to Meet Society's Needs*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=9829. Last accessed August 20, 2010.

[4] NRC, 1997, *The Evolution of Untethered Communications*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=5968. Last accessed August 20, 2010.

Commercial Software Supply Chains

The DoD's supply chain has become increasingly complex. One driver has been the commercial success of software ecosystems that provide a foundation on which to build defense systems. Another driver has been the increasing use of tools for collaboration at a distance. These are the same technologies, infrastructure, and practices that have enabled the globalization of diverse services. (Indeed because of this it is easy to erroneously conflate supply-chain richness with the globalization phenomenon.)

The factors driving the supply-chain structure include not only the direct costs of development, but also the resulting management agility (such as the ability to revisit choices in infrastructure, technology, and particular suppliers) and rapid access to specialized expertise (domain knowledge and requirements, code development, vendor components, testing and evaluation, process structuring, software architecture, and so on). For commercial applications, these factors combine to enable large firms to quickly adapt and enhance their business models to address competitive challenges. The DoD can and should realize similar advantages, but of course it also needs to address the risks, including both the sourcing risks intrinsic in this kind of supply structure and the particular requirements risks that derive from the defense mission.

The complexity of software supply-chain structures is evident in diverse sectors. A single firm in a sector such as financial services, health care, or manufacturing may develop software at dozens of

separate sites around the world. This software is likely to depend on infrastructural components from dozens of vendors, each of whom may also have global operations. This is how the advantages of cost, agility, and expertise are realized and fundamental new functionalities are achieved, for example, those involving cross-cutting capabilities related to business intelligence or enterprise process management (both of which are highly relevant to the defense mission). But the risks must also be addressed—when the software components are interconnected within a single system, all of these components may be "behind the firewall" and with direct access to mission data. (Issues related to assurance, including challenges related to supply chains, are addressed later in this report, in Chapter 4.)

The market forces that drive the complexity of the modern supply-chain structure for software systems have long been present, but the supply chain and ecosystem richness are only relatively recent phenomena, enabled by the richness of modern software technology. The enabling technologies include modern programming languages, system software components, network protocol development, object-oriented frameworks, emerging service-oriented concepts (e.g., cloud, SAAS, SOA), advanced tooling for team development and collaboration, and process support. Indeed, a number of these core ideas are results of DARPA and Service funding of research projects in prior decades.

An important element of the globalization phenomenon is the pace at which global suppliers outside the United States, in many countries, are moving up the value chain—that is accounting for an increasing share of the overall value embodied in a product or service. Global suppliers, which in the early days focused primarily on low-technology offerings such as providing black-box testing services for Web-based software systems developed in the United States and elsewhere or on provisioning remote first-tier technical support capability, are now developing the software for those systems directly, as well as engaging in requirements analysis, architecture, and design for those systems. This commercial trend, accelerated through direct strategic investment by governments, exacerbates the DoD concerns about the mission impact of foreign influence on DoD software—namely, the risk of unwanted functionality in delivered software (and hardware as well, where the assurance challenges can be greater).

Finding 1-3: The DoD relies fundamentally on mainstream commercial components, supply chains, and software ecosystems for both business systems and many mission systems. Nonetheless, the DoD has special needs in its mission systems driven by the growing role of software in systems. As a result, the DoD needs to address directly the challenge of building on and, where appropriate, contributing to the development of mainstream software that can contribute to its mission.

DoD Software Supply Chains

The DoD is aggressively applying these ideas for business applications. In the case of IT systems and components such as databases, operating systems, and business systems applications, the DoD can align well with commercial products being produced to support industry.

At the same time, however, software has become a critical differentiator in most mission-related systems and services—and (as noted above) it is growing in the extent and depth of its impact every year. It is safe to claim, in fact, that the largest opportunities for successful differentiation in new mission systems are very often derived from software-manifest capabilities. It is therefore not surprising that the largest risks in systems development are associated with the software production.

Almost always, DoD mission systems rely on a combination of innovative functionality and capabilities already present in established ecosystems. The DoD must obviously leverage the extensive commercial development of software processes, methods, tools, architectures, and products. But, as the committee notes below, the DoD must also take action, as it has done historically, to foster the capabilities of its supply chain, broadly construed, to enable it to stay ahead of its rapidly advancing adversaries. In particular, this new reality poses challenges not only for developing innovative functionalities, but also for assurance and ecosystems leadership.

As noted above, larger-scale innovative mission-focused applications generally include both innova-

tive custom software components and components based on existing ecosystems. As a result, the DoD necessarily relies to some extent on the extant supply-chain structures with their diversity of suppliers. This creates both technical and management risks related to assuring control over functionality and assurance.[24] These challenges focus on understanding the extent and nature of supply-chain and ecosystem dependencies, on developing both incentives and technical means to mitigate assurance risks, and on understanding the leadership challenges with respect to sustaining some control (or at least awareness) over future trajectories where there are necessary ecosystem dependencies.

Governments are recognizing the difference between component-level participation in an ecosystem-associated supply chain and architectural leadership of that ecosystem.[25] U.S. firms have developed and led the evolution of most of the key ecosystems on which the DoD and the entire industry rely. This is a consequence of technological and market leadership, and the technological aspects of this leadership, in turn, are in large measure consequences of the long record of R&D investment in core computer and information technologies by DARPA and the Services and a small number of other Networking and Information Technology Research and Development (NITRD) agencies, principally the National Science Foundation (NSF) and, historically, NASA. These investments are now diminished (Box 1.5). The committee considers priorities regarding this investment (Chapter 5) as well as arguments for and against a scaling up of this investment (this chapter) and means by which the investment can be evaluated and optimized (Chapter 5).

Summary—Software and the DoD

Software is highly significant for the DoD and becoming more so. The DoD depends not only on the ability to develop new code, but also on commercial software capability, particularly as manifested in established evolving software ecosystems. Software supply chains are growing in scale, complexity, and geography, and the influence of these shifts on DoD software must be considered.

Although the United States continues to retain innovation leadership in software areas important to the DoD, there are three proximate factors that could cause the loss of that leadership. First, as noted above and documented in Box 1.5, the DoD investment in software producibility has in recent years diminished considerably from its prior levels, which had been sustained for more than three decades. Second, concomitant with the diminishing of U.S. investment is a ramping up of investment by foreign governments in their national IT capabilities, including in software.[26] The third factor, also as noted above, is the inexorable trend of globalization and rich supply chains.

Of course, very strong shifts overseas have happened in other sectors, such as consumer electronics, and there is still debate regarding the strategic impact of these shifts. It is the committee's view, however, that the leveraged role of software and the particular special role of software in defense and national

[24] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[25] "Internet—New Shot in the Arm for US Hegemony," *China Daily*, January 22, 2010. Available online at http://www.chinadaily.cn/china/2010-01/22/content_9364327_3.htm. Last accessed August 20, 2010.

[26] See Organisation for Economic Co-Operation and Development (OECD), *Information Technology Outlook 2006*, Paris, France: OECD. Available online at http://www.oecd.org/document/10/0,3343,en_2649_37441_37486858_1_1_1_37441,00.html#TOCat. Accessed February 26, 2008. Also see NRC, 2009, *Assessing the Impacts of Changes in the Information Technology R&D Ecosystem: Retaining Leadership in an Increasingly Global Environment*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12174. Last accessed August 20, 2010. See also Ashish Arora and Alfonso Gambardella, eds., 2005, *From Underdogs to Tigers: The Rise and Growth of the Software Industry in Brazil, China, India, Ireland, and Israel*, Oxford, England: Oxford University Press, pp. 171-206. Rafiq Dossani and Martin Kenney, 2007, "The Evolving Indian Offshore Services Environment: Greater Scale, Scope and Sophistication," *Sloan Industry Studies Working Papers*, Number WP-2007-34, 2007. Available at http://www.industry.sloan.org/industrystudies/workingpapers/index.php. Accessed February 26, 2008. OECD, 2006, "China Will Become World's Second Highest Investor in R&D by End of 2006, Finds OECD," OECD Online, http://www.oecd.org/document/26/0,2340,en_2649_201185_37770522_1_1_1_1,00.html. Accessed February 26, 2008.

BOX 1.5
The Decline in Federal Investment in Software Producibility Research

The extent and structure of federal IT-related R&D investment are documented in a series of reports published by the National Coordination Office (NCO) for Networking and Information Technology Research and Development (NITRD). These reports have been issued annually since the first such report was included in the FY1992 federal budget submission, and they document research sponsorship from a diverse set of federal agencies related to IT and networking. NITRD is a multi-agency coordination activity, which operates under the auspices of the White House Office of Science and Technology Policy (OSTP) and the Office of Management and Budget (OMB), but the funding that is reported is in the individual agencies' budgets and generally under their control. The reports include extensive narrative descriptions of research accomplishments and plans, as well as a budget matrix that shows investment levels by agency, organized into a set of eight categories.[1] The budget matrix shows both proposed amounts for the forthcoming fiscal year and approximate actual amounts for the then-current fiscal year.

There are two categories that relate to software producibility—Software Design and Productivity (SDP) and High Confidence Software and Systems (HCSS). The committee analyzed the trends in these two categories over the past decade and related its findings to the overall NITRD-coordinated budget that totals investment in all eight categories. (Note: The analysis excluded National Institutes of Health (NIH) data. This was done for two reasons: First, NIH changed its reporting methodology in 2010, which creates non-commensurability for a longitudinal analysis. Second, NIH allocations among the NITRD topic categories were determined through the use of an automated text-based pattern-matching algorithm. The committee believes this approach, particularly in topics related to software production generally (rather than, for example, the production of software for particular applications), is likely to lead to significant over-reporting of application software development projects as SDP or HCSS research.)

The principal result of the analysis is that the SDP and HCSS investments, separately and combined and in absolute dollars and as a percentage of the NITRD budget, have dropped considerably in the past 5 years. At the same time, the overall NITRD-coordinated budget has grown. For example, from 2004 to 2010 the combined allocation to SDP and HCSS fell by 45 percent, while the overall NITRD budget more than doubled. On a percentage basis, the combined SDP and HCSS allocation fell by a factor of almost four, from 24.6 percent of the NITRD total in 2004 to just 6.5 percent of the total in 2010.

One of the challenges in this type of budget analysis is the breadth of the categories and the imprecision of category boundaries. This challenge is unavoidable in the analysis of research budgets, but it is particularly difficult in the analysis of NITRD budgets because the different agency staff may apply slightly different criteria when categorizing diverse and innovative research projects. When category labels change, for example, with the introduction of the category of Cybersecurity and Information Assurance (CSIA), it is very likely that some projects in HCSS were relabeled as CSIA. Although it would be desirable to assess each grant for its relevance to the categories or, better, to the particular technical disciplines that support "software producibility," this would be infeasible because of the large number of research grants and contracts and also because agencies are reluctant to share detailed data regarding awards and category assignments. An analysis of the narrative descriptions associated with the categories in the NITRD report suggests that there is an acceptably close alignment of SDP and HCSS with the overall investment that might directly relate to software producibility. The narrative descriptions do reveal some areas included in SDP or HCSS that might not be included in a "software producibility" category, for example, due to application specificity or other attribute.

Taking all this into consideration, the committee judges that the combination of SDP and HCSS is sufficiently close to a notional category of software producibility that we accept it as a surrogate for overall investment across NITRD agencies (NIH excluded) in research that relates to the present report.

[1] The categories have slowly evolved over the years, but categories related to software have remained unchanged for the past decade.

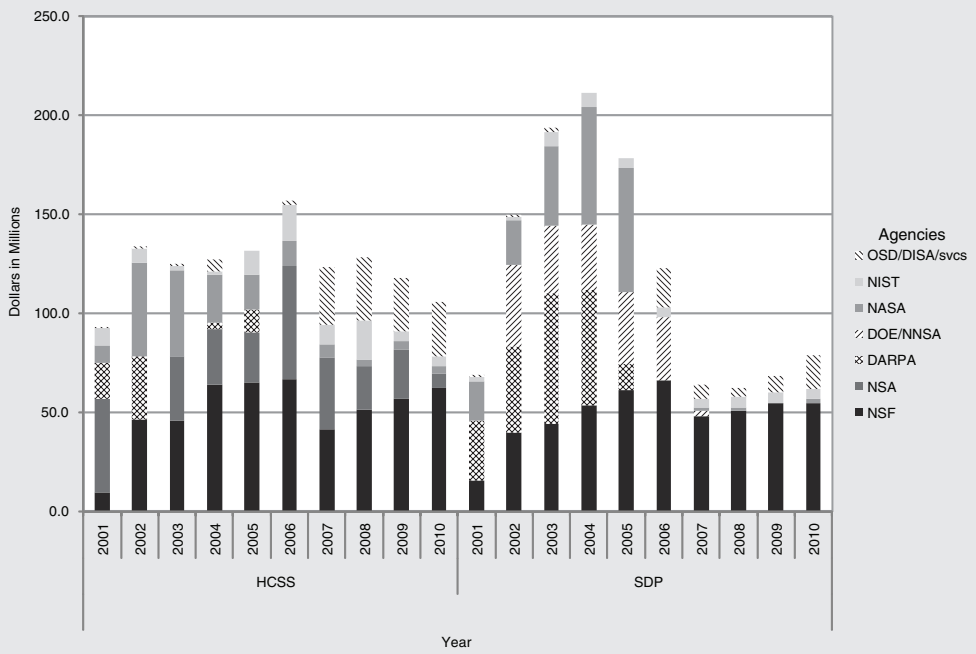

Figure 1.5.1 Investment in HCSS and SDP by agency and by year. NOTE: Office of the Secretary of Defense (OSD), Defense Information Systems Agency (DISA), and Service investments have been rolled up into a single category that covers defense agency investments other than those in DARPA and the National Security Agency (NSA). NIH amounts excluded for reasons noted in the text.

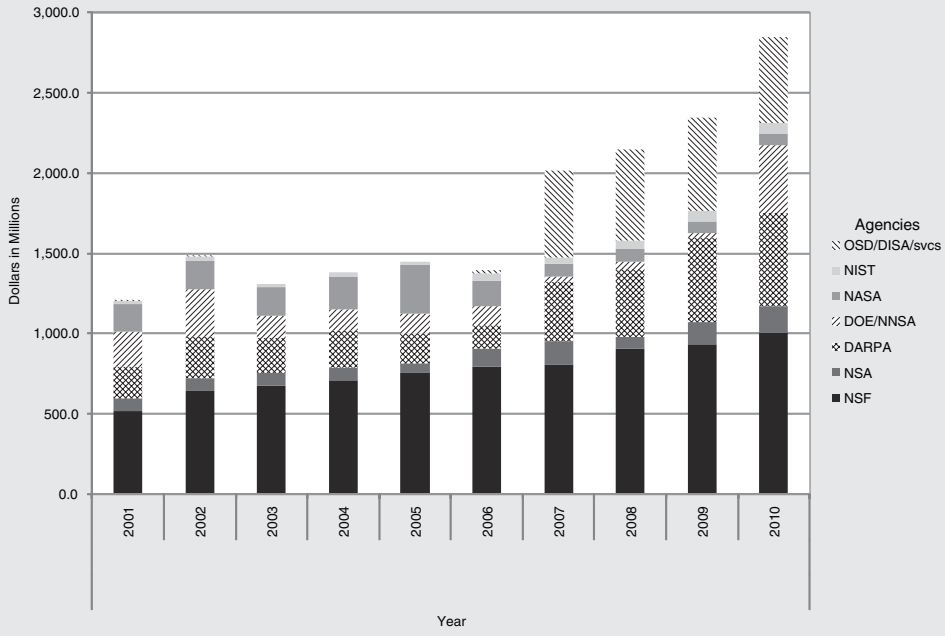

Figure 1.5.2 Total NITRD investment by agency and by year. NOTE: OSD, DISA, and Service investments have been rolled up into a single category that covers defense agency investments other than those in DARPA and NSA. NIH amounts excluded for reasons noted in the text.

BOX 1-5 Continued

More details regarding the state of research investment are shown in Figures 1.5.1 and 1.5.2. The former shows the investment by year and by agency for HCSS (left side) and SDP (right side). It is evident from Figure 1.5.1, for example, that NASA, DARPA, and the Department of Energy (DOE) have stepped almost completely away from engaging in research related to SDP. Figure 1.5.2 shows the overall extent of the NITRD investment in all eight categories of research related to networking and IT.

Two trends are immediately apparent from these charts: there was a surge of SDP and HCSS investment in the mid-2000s by several agencies—DARPA, DOE/National Nuclear Security Administration (NNSA), and NASA—followed by a precipitous drop in their investments and in the total investment. The result is that NSF is now the dominant source of investment in both categories. Figure 1.5.3 shows the percentage of total NITRD investment in SDP and HCSS. It illustrates that while the total NITRD investment more than doubled over the past decade, the percentage of investment in SDP and HCSS fell off sharply.

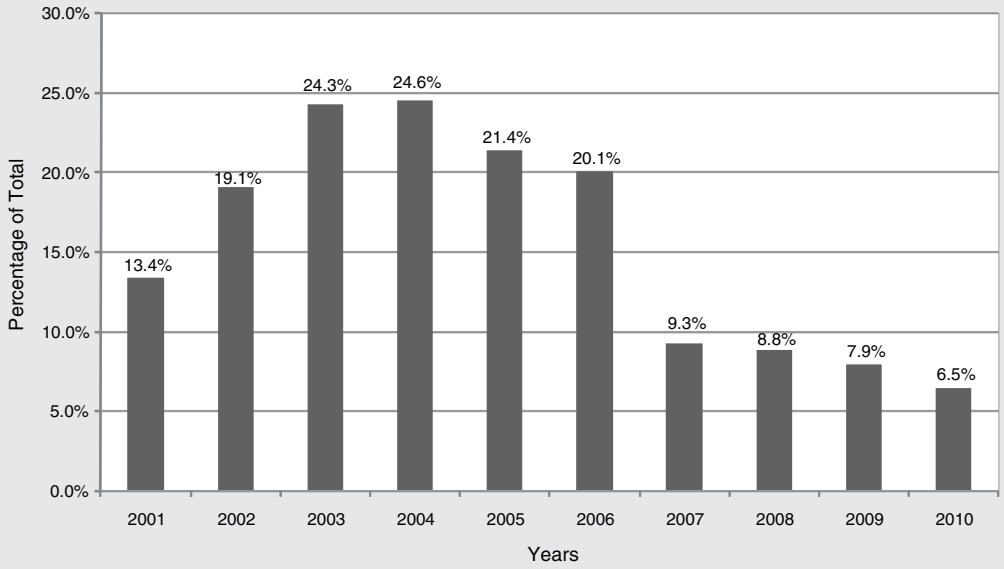

Figure 1.5.3 Percentage of total NITRD investment in either SDP or HCSS.

security systems of all kinds make this kind of shift much more consequential for defense software producibility and for U.S. ability to advance overall defense system capability.

In exploring the role of the DoD in advancing software producibility, which is the topic of Chapters 2, 3, and 4, the committee considers the interplay of four factors:

1. *Productivity.* The DoD benefits from efficiencies gained in the development of innovative functionality through advances in mainstream software producibility. For mission systems, there are particular challenges relating to requirements and validation, architecture and modeling, process and measurement, and tools and language systems.
2. *Innovation.* The DoD relies on technologically enabled advances in software producibility to enable the more effective creation of unprecedented systems and the interconnecting of existing and new systems to deliver advanced functionality with acceptable cost and risks.

3. *Assurance.* The DoD faces new challenges to addressing the risks associated with diverse and international supply chains, including the development of practices and technologies for software assurance.

4. *Ecosystems and infrastructure.* The DoD unavoidably relies on mainstream software ecosystems in defense systems and therefore has a stake in the processes by which those ecosystems evolve and are led.

THE ROLE OF THE DOD IN ADDRESSING ITS SOFTWARE NEEDS

Given the importance of software to the DoD, and to its mission systems in particular, and given also the ongoing rapid advances in software capability worldwide, it is vital to ensure that the department can not only meet its software needs now, but also sustain its software leadership well into the future. A key question addressed by the committee is to what extent the DoD, without providing its own explicit R&D stimulus, can rely on industry—specifically the domestic defense industrial base and supporting vendors—to produce software innovations in areas of defense significance at a rate fast enough to allow the DoD to fully meet software requirements and remain ahead of potential adversaries. This leadership must be with respect to both the capability of systems and effective defense against attacks on those systems.

As noted above, the DoD has particular requirements that must be dealt with on systems that are both very large scale and have life-critical mission requirements. Although these areas may overlap with civil and commercial needs, very often the DoD requirements are more sophisticated and cutting-edge than those in the rest of the marketplace. Also, DoD adversaries may choose to "attack" software in the supply chain during development phases of a project—security is much more than about attacks staged over networks during system operations. Additionally, major DoD development projects are structured in a way that often keeps development teams at arm's length from the key operational mission stakeholders and from overall project management. For these reasons, technological advancement would significantly benefit the DoD's ability to produce the software it needs. The areas identified in this report, and particularly in Chapter 5, are areas where the committee sees the DoD as having leading demand. The committee notes that the issue is not areas where the DoD has "unique" requirements, but rather the much broader category of areas where it has *leading demand* with respect to particular kinds of requirements. One obvious example is software assurance, where DoD and national security needs may go well beyond even what is being developed for commercial financial services or health care devices. Another example is the risk-managed development of unprecedented architectural design of the kind required to create high-interconnectivity systems such as FCS, net-centric systems, and many other major defense platforms that have few commercial precedents or analogs.

The committee notes that even where industry is aggressively innovative, it may not have sufficient incentives to produce the technology and supporting tools necessary to generalize application-specific software innovations. Additionally, the technologies may manifest innovative concepts, but in a way that cannot be readily adopted by the DoD, for example due to the safety, reliability, and assurance considerations particular to defense applications that, to be addressed, require further technological innovation.

It would thus be overly optimistic to conclude that DoD needs such as these will somehow be adequately addressed through a combination of demand-pull from the DoD and technology-push from the defense sector (i.e., firms that primarily supply the DoD) or the broader commercial IT sector. In many other industries and infrastructures, this may be a legitimate conclusion, and in these areas the best policy may be for the DoD to follow the market. However, this is not generally true for software technology, particularly as needed for defense mission systems, where the DoD has leading demand in multiple critical areas, as detailed later in this report.

Finding 1-4: The DoD's needs will not be sufficiently met through a combination of demand-pull from the military and technology-push from the defense or commercial IT sectors. The DoD cannot rely on industry alone to address the long-term software challenges particular to defense.

The above finding is based on consideration of both the history of software innovation and the set of future needs identified in this study.

Commercial R&D Investment

Defense contractors do invest extensively in software research, but generally speaking it is focused on manifesting specific capabilities in supporting the competitiveness of bids through differentiated skills and products. Commercial IT firms also invest in software research and form an important part of the defense IT supply chain, but may not necessarily carry out research aimed at meeting the DoD's needs. Both defense and commercial IT firms lack strong incentives to invest directly in broad-impact areas such as these, particularly when many of the advantages derived from the investment are non-appropriable, in that the associated intellectual property cannot be readily controlled and as a consequence the benefits may readily diffuse not only into the supply chain for the contractor but also to competitors (more on this point is given below).[27] Indeed, it is important to note that there are certain technological improvements that may in fact not necessarily be good for business, even when the DoD derives capability and acquisition advantage.

Additionally, and perhaps most importantly, there is the issue of horizon. Prudent business decisions are generally informed by return-on-investment calculations, which depend on (1) appropriability, (2) timeliness, (3) investment risk, and (4) measurability/observability of return. Many improvements in practices, for example, come only after sustained commitments and much technical exploration (after which benefits may rapidly diffuse across the industry). Additionally, benefits, even when judged significant by technical leaders, may be difficult to quantify, due to the measurement challenges that persist in software and the software-related aspects of systems engineering (see Chapter 2). In other words, in investing in software producibility, all four of the elements above are problematic. And therefore, in a competitive market, individual companies generally have few market incentives for such investments. These factors tend to drive internal R&D investments in contractors toward a combination of addressing business needs expected 1 to 2 years in the future and avoiding technological surprise from competitors. Government research and development in software producibility, from a purely structural perspective, can be less focused on appropriability and investment risk. Additionally, government can invest directly in improving measurement capability, when that is a source of risk (see the discussion of measurement in Chapter 5).

The Challenge of ROI and Appropriability

It is an economic reality that it is difficult for a firm to make a compelling case of return on investment (ROI) to undertake innovations when those innovations have a non-appropriable character, which is to say that the intellectual property associated with the innovations diffuses broadly into the engineering discipline and the economy. This is a familiar issue to those involved in defining industry-wide best practices, standards, and other commonalities. Many of the most important and highly leveraged, government-originated innovations (undertaken by both academia and industry) are in the economic "commons." This creates challenges in measuring value created, because the value has broad and dif-

[27] See, e.g., NRC, 2002, *Information Technology, Research, Innovation, and E-Government*, Washington, DC: National Academies Press, p. 101. Available online at http://www.nap.edu/catalog.php?record_id=10355. Last accessed August 20, 2010.

fuse benefits.[28] A recent NRC report on software and the economy[29] notes that "the economic rationale for government investment is based on the non-appropriability of many significant IT innovations, including the most widely used idiomatic data structures and algorithms, as well as design and architectural patterns. Moreover, the IT industry relies on a number of technical and process commonalities or standards such as the suite of Internet protocols, programming languages, core design patterns, and architectural styles." These innovations effectively "raise everyone's boat" in the same way as do government investments in bioscience, health care, and other strategically important scientific disciplines. This includes many of the most highly leveraged areas of software research such as improvements in abstraction mechanisms, design notations, programming languages, software analysis and model checking, basic algorithms, design patterns and architecture concepts, and other core techniques. This is not to say that industry does not invest in non-appropriable results—it does so extensively in the area of standards and also through investment in university projects and other activities. But (for the reasons cited here) the incentives to sustain this investment are lower than for proprietary development efforts.

The committee notes, in addition, that when research results are not appropriable, researchers are less likely to be able to secure patents. (Thus, it can be safely hypothesized that direct revenues from licensing university-owned patents are likely to be significant underestimates of the value created by federally funded research, especially in the case of software-related university inventions.)

To complicate matters further, the manner in which software is protected as intellectual property is often distinct from what is done in other fields such as biomedicine. A software system is often a combination of differently protected elements, organized following architectural elements of several ecosystems along with custom elements. There is often considerable intellectual property embedded in the architectural designs themselves, even in the absence of components that populate those designs. All this combines to make the non-appropriable, yet valuable, aspects of the work hard to identify.

Finally, the committee notes that the software industry is shifting from the development of software products to an increased focus on integration, custom development, and other services.[30] More than half the revenues of software product companies are now coming from services rather than from product sales, but product sales tend to be the most scalable and profitable part of the business.[31] Additionally, the software-product sector is consolidating and thus shrinking in numbers, decreasing from more than 400 to less than 150 publicly listed software product companies on U.S. stock exchanges in the past 8 years.[32] Another consequence of the shifts toward services and consolidation may be a reduction in

[28] Indeed, fear of value diffusing to competitors that can assimilate it more efficiently into their practices can create a negative incentive to invest, although it can be dangerous to make this judgment when evidence is lacking, because innovation leadership may be less easy to recover once lost.

[29] NRC, 2006, *Measuring and Sustaining the New Economy, Software, Growth, and the Future of the U.S Economy: Report of a Symposium*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/openbook.php?record_id=11587. Last accessed August 20, 2010.

[30] For quite a few years, about two-thirds of global revenues in the software industry have actually been from services (such as custom software development, maintenance, IT consulting, and technical support), and only one-third of revenues have come from the product companies. See Michael A. Cusumano, 2004, *The Business of Software,* New York: Free Press, p. 46, footnote 19, citing Standard & Poor's annual data. An issue in these analyses is how the word "services" is defined—it is used both for custom development/integration by teams of people and also for "software-as-a-service" (SAAS) and cloud-based delivery of software value. Additionally, the distinction between "product" and "service" in software is becoming increasingly muddy as licensed product software delivered to customers is complemented by off-site SAAS and cloud-based software generally. For many end users, for example, there may be relatively little distinction in the experience, say, of Microsoft Office tools on a desktop computer (a "product") and Microsoft Office Live or Google documents tools on a browser (a "service"). A similar statement could be made at the enterprise level, for example, regarding customer relationship management (CRM) software. This shift is the result of evolving business models enabled by technology and infrastructure developments.

[31] Thus, only about one-sixth of global software industry revenues (half of the one-third of revenues from products) are from product sales. See Michael A. Cusumano, 2008, "The Changing Software Business; Moving from Products to Services," *IEEE Computer* 40(1):20-27.

[32] See p. 22 in Michael A. Cusumano, 2008, "The Changing Software Business; Moving from Products to Services," *IEEE Computer* 40(1):20-27. This may also reflect post-bubble consolidation in both technology- and media-focused companies.

support for new software research. De facto consortia, often but not always manifest as open-source projects, are often the centers for industry-wide innovation.[33]

Changes in the DoD Innovation Environment

The question regarding the extent to which the DoD can meet its ongoing software needs through innovation-push from the commercial sector has to be answered in the context of three additional factors, which follow three significant shifts in the environment of technology innovation.

The first is the growing globalization of the software industry, as noted earlier, with rapid gains in capability in India and China (and elsewhere in Asia) and Russia as well as steady gains in Europe.[34,35] This shift creates competitive pressures with respect to the rapidly increasing proportion of defense mission capability embodied in software. It also amplifies the challenge of mission assurance, given the increasing extent to which DoD software will likely be developed in foreign countries. This topic was taken up by the committee, and Chapter 4 addresses practice and technology issues related to software assurance, including both preventive measures in the engineering process and evaluative measures appropriate for development and for test and evaluation. Other dimensions of cybersecurity are consequential but not within the scope of this report—these were discussed at length in the 2007 DSB Task Force report *Mission Impact of Foreign Influence on DoD Software*.[36]

The second shift has been the reduction over the past decade of direct DoD investment in advancing software capability within the defense industrial base and its supply chain (see Box 1.5). Although this shift may be under reconsideration, it nonetheless raises a key question that was considered extensively by this committee, which is whether industry, without explicit R&D stimulus from the DoD, will produce innovations in areas of interest to the DoD of the kind and extent that are needed to meet the ongoing rapid growth in DoD software requirements.

The third shift is a consequence of the second, which is the reduction in PhD output due to the drop in R&D investment in software producibility. Historically, the DoD has had a significant leadership role in creating and sustaining the innovation advantage of the United States in IT and in fostering new generations of innovators and technical leaders in computer science and IT. This role has been evident from the earliest days of computing during World War II. That this DoD investment has been a significant driver of U.S. capability and innovation in information technologies is documented in several national studies.[37,38,39] This has had the salutary benefit of enabling the United States to develop and retain innovation and ecosystems leadership. In areas related to software producibility, including high-

[33] Examples in the realm of open source include Apache, Linux, Eclipse, and other widely adopted open source. In addition, formal consortia are often created to address industry-wide issues, as in the case of W3C (HTTP, HTML, XML, CSS, and other web-related standards) and TCG (TPM, trusted storage, and other trust-related standards for hardware). Finally, expert groups are often convened by standards organizations to address common issues, as in the case of JPEG (ISO and ITU) and MPEG (ISO).

[34] See Michael A. Cusumano, 2006, "Where Does Russia Fit into the Global Software Industry?" *Communications of the ACM* 49(2):31-34. See also, Michael A. Cusumano, 2006, "Envisioning the Future of India's Software Services Business," *Communications of the ACM* 49(10):15-17.

[35] In China, there are private and state-connected companies under government sponsorship to develop ecosystems and infrastructure software (China versions of CDMA/GSM, embedded operating systems, and search engines, for example) to reduce dependence on firms such as Qualcomm, Nokia, Microsoft, Google, etc.

[36] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[37] NRC, 1997, *Ada and Beyond: Software Policies for the Department of Defense*, Washingon, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=5463. Last accessed August 20, 2010.

[38] NRC, 2000, *Making IT Better: Expanding Information Technology Research to Meet Society's Needs*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=9829. Last accessed August 20, 2010.

[39] NRC, 1997, *The Evolution of Untethered Communications*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=5968. Last accessed August 20, 2010.

confidence systems, data from the NITRD coordination reports show that the DoD has reduced its basic research investment (see Box 1.5).

In addition to these shifts, there may also be cases where industry has little economic incentive to acknowledge fundamental gaps in knowledge. This is not so unusual. One obvious example is ill-structured contracts, often from the government, that create perverse incentives—for example, where project difficulties (or inadequate tooling and practices) may accrue to a contractor's economic benefit in the form of increased contract costs and profits, along with long-term revenue streams resulting from costly post-deployment repair and maintenance requirements. A second example is the consequence of poor measurement capability (as noted above), particularly relating to quality and security—"assurance metrics" in the terminology of the 2007 DSB Task Force report.[40] When metrics and observables are lacking, it is difficult to construct a business case for improvement of the underlying phenomena of concern—quality and security in this case. A third example is inadequate best practices. When metrics are weak, we must rely disproportionately on folklore-derived best practices and processes and organizational maturity to achieve product-related goals. Process compliance is relatively easier to achieve and certify than quality, but in software it is not always strongly correlated. As noted, fundamental improvements in best practices to enhance what can be achieved in terms of systems capability, productivity, quality, agility, and other characteristics sometimes fall into the category of non-appropriable innovations, discussed above, and thus may not readily be the subject of industry investment. These factors combine to lower industry economic incentives to address the producibility challenge.

Recommendation 1-1: The DoD, through its Director of Research & Engineering (DDR&E), should regularly undertake an identification of areas of technological need related to software producibility where the DoD has "leading demand" and where accelerated progress is needed to support the defense mission.

THE NECESSITY OF INNOVATION IN SOFTWARE

Is There a Need to Innovate?

That global suppliers are moving up the value chain[41] suggests the possibility that U.S. leadership may be eclipsed in many of the core technologies related to systems architecture, languages and tools, and software assurance, as well as with respect to key design elements of software ecosystems and infrastructure. This suggests a key question, particularly as we contemplate the commitment of resources to new R&D activity: Is there, in fact, strategic value in retaining U.S. leadership in software producibility? The committee argues strongly in the affirmative, based on the unique role and technological characteristics of software.

The DSB Task Force report on foreign software also asks this question, focused on the particular area of software assurance, and offers an affirmative response, noting the essential requirement that the United States maintain advanced capability for "test and evaluation" of IT products. In other words, reputation-based or trust-based credentialing of software ("provenance") needs more and more to be augmented by direct, artifact-focused means to support acceptance evaluation. The DSB Task Force recommends more effective direct evaluation by consuming organizations throughout the software supply chain, including better ways for producers to create software for which direct evidence of critical quality

[40] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[41] This is in the sense of moving from more routine activities that require minimal technological sophistication (such as first-tier call centers, system-level "black-box" testing focused on the user experience, and similar activities) to more value-enhancing activities (such as custom software design and development). For software, this shift is enabled primarily through increased sophistication in technology and business practices.

and functionality attributes can be provided. It concluded that test and evaluation should be supported by a broad range of software engineering technologies and interventions, not just those employed at the late test phase of development. (These issues are explored more extensively in Chapter 4.)

Both the 2007 DSB Task Force report on foreign software[42] and the DSB 2000 report on defense software[43] also highlight the importance of commercial technology to the DoD, including the essential elements (operating systems, databases, application servers, and so on) of most of the predominant software ecosystems architectures. DoD's historical investment in basic research has influenced these commercial technologies in ways that have facilitated DoD adoption (for example, early attention to scalability, process separation, interconnection, and survivability in operating systems and distributed systems infrastructure). Without continued research investment, that influence will diminish just when the off-the-shelf technologies are rising in importance to DoD systems.

Additionally—and looking beyond software assurance into the other critical dimension of software producibility—without continued research investment, the DoD will lose effectiveness in its ability to undertake custom software engineering to rapidly achieve high levels of capability and to adapt with maximum agility to changes in the operating environment. A significant loss of U.S. leadership in either area could threaten the DoD's ability to produce and assure the software it requires.

Historically, the DoD investment has been an enabler, both directly and indirectly, of U.S. technological leadership in software innovation (Box 1.6). This has helped enable the United States to be a leader in the development of software ecosystems, which means that the DoD is able to rely on ecosystem architectures and components with greater confidence. It has afforded a high level of capability in software producibility for the necessary custom software that the DoD must develop or acquire.

Will Software Capability Reach a Plateau?

Some have contended that software capability may be reaching a plateau, and as a consequence there is reduced need for leadership and innovation, because the technologies are inevitably commoditizing and the engineering focus is shifting to optimization and routinization. This suggestion is sometimes offered as an analogy with other technology disciplines, ranging from many specialized materials to display panels and memory chips. The committee views this contention as dangerously incorrect.

The reality is that software is not at a plateau, despite the fact that suggestions of the possibility are made on a regular basis. Consider, for example, the ambitious aspirations of those developing Fortran (Box 1.7). A similar story can be told about the "Fourth Generation" database languages introduced a few decades later and, more recently, about languages for business rules. These are all major innovations with far-reaching impact, and in all cases they delivered considerable value. But they also inspired computer users to greater ambitions, and thus the limits of these innovations were reached.

One of the most significant special characteristics of software, as noted at the outset of this report, is its unboundedness—the lack of natural physical limits on its scale and complexity. As the sophistication of languages, models, tools, and practices increases, the ambitions of computer users continue to be realized. These characteristics also mean that while these developments move us forward, they do not actually get us closer to "being there" at some plateau of capability and emerging commodity status. New software-manifest capabilities are constantly emerging—for example, machine-learning technology is now used in applications ranging from data mining to robot design and quality-of-life enhancement for seniors. The profound fact is that software seems to be limitless. For software, "continuous improvement" in capability (as distinct from process) is less a matter of fine-tuning than an innovation

[42] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[43] DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA385923. Last accessed August 20, 2010.

BOX 1.6
Lessons About the Nature of Research in IT

The following material is reprinted from National Research Council, 2003, *Innovation in Information Technology,* National Academies Press, Washington, DC, pp. 2-4.

The Results of Research

—America's international leadership in IT—leadership that is vital to the nation—springs from a deep tradition of research. . . .

—The unanticipated results of research are often as important as the anticipated results. . . .

—The interaction of research ideas multiplies their impact—for example, concurrent research programs targeted at integrated circuit design, computer graphics, networking, and workstation-based computing strongly reinforced and amplified one another. . . .

Research as a Partnership

—The success of the IT research enterprise reflects a complex partnership among government, industry, and universities. . . .

—The federal government has had and will continue to have an essential role in sponsoring fundamental research in IT—largely university-based—because it does what industry does not and cannot do. . . . Industrial and governmental investments in research reflect different motivations, resulting in differences in style, focus, and time horizon. . . .

—Companies have little incentive to invest significantly in activities whose benefits will spread quickly to their rivals. . . . Fundamental research often falls into this category. . . . the vast majority of corporate R&D addresses product and process development. . . .

—Government funding for research has leveraged the effective decision making of visionary program managers and program office directors from the research community, empowering them to take risks in designing programs and selecting grantees. . . . Government sponsorship of research especially in universities also helps to develop the IT talent used by industry, universities, and other parts of the economy. . . .

The Economic Payoff of Research

—Past returns on federal investments in IT research have been extraordinary for both U.S. society and the U.S. economy. . . . The transformative effects of IT grow as innovations build on one another and as user know-how compounds. Priming that pump for tomorrow is today's challenge.

—When companies create products using the ideas and workforce that result from federally sponsored research, they repay the nation in jobs, tax revenues, productivity increases, and world leadership. . . .

process that regularly delivers order-of-magnitude jumps in capability. The improvements are not just outcomes from the Moore's Law curves and other exponentials in the underlying physical infrastructure of processors, memory, and networks. Those improvements are significant enablers, but our ability to produce increasingly complex software artifacts has largely been enabled by a steady pace of technological breakthroughs in software practices, technology, languages, models, and tooling. (The *performance* of software has been enabled by the development of new algorithms, by advances in hardware, and by simplification and rescoping of the computational problems being solved.)

Similar observations apply to software ecosystems. These are a critical and essential development to provide conventionalized access to infrastructure and capability on which systems are built. They are, in fact, the success of software reuse.[44] As technical progress is made, these structures evolve, with increasing levels of infrastructure capability, for example in operating systems, databases, application servers, frameworks of various kinds, data center services, and so on. In a sense, this is analogous to a

[44] Butler W. Lampson, 1999, "Software Components: Only the Giants Survive," *21st International Conference on Software Engineering (ICSE'99),* Keynote Address. Available online at http://research.microsoft.com/en-us/um/people/blampson/70-softwarecomponents/70-softwarecomponents.doc. Last accessed August 20, 2010.

BOX 1.7
Fortran Was a Breakthough but Did Not Lead to a "Software Plateau"

It is instructive to consider the publication more than 50 years ago of the 1958 landmark paper by John Backus describing the first Fortran compiler.[1] The title made reference to "automatic programming," and indeed this phrase was widely used at the time, including in the titles of technical conferences.[2] The point of this phrase, with respect to Backus's great accomplishment, is that there was a much more direct correspondence between his high-level programming notation—the earliest Fortran code—and pure mathematical thinking than had been the case with the early machine-level code. One can construe that it was imagined that Fortran enabled mathematicians to express their thoughts directly to computers, seemingly without the intervention of programmers. This was an extraordinary and historical breakthrough. But we know that, in the end, those mathematicians of 50 years ago soon evolved into programmers as a consequence of their growing ambitions for computing applications. Just a decade after the Backus paper, Fortran was used to support list-processing applications, typesetting applications, compilers for other languages, and other applications whose abstractions required some considerable programming sophistication (and representational gerrymandering) to be represented effectively as Fortran data structures—arrays and numeric values. (See further discussion in footnote 26 in Chapter 5.)

[1] John W. Backus, November 1958, "Automatic Programming: Properties and Performance of FORTRAN Systems I and II," in *Proceedings of the Symposium on the Mechanisation of Thought Processes,* Teddington, Middlesex, England: The National Physical Laboratory. Available online at http://archive.computerhistory.org/resources/text/Fortran/102663114.05.01.acc.pdf. Last accessed August 10, 2010.

[2] An early example is the 1954 "Symposium on Automatic Programming for Digital Computers." See John Backus, 1978, "The History of Fortran I, II, and III," *History of Programming Languages*, New York: ACM.

process of commoditization, in that many of the key architectural interfaces effectively define market structures for competitive supply of capabilities.

But, at the same time, providers and clients continue to innovate above and around the infrastructure, creating new kinds of capability and differentiation. Additionally, there is very often continued innovation within the infrastructure to add capability, create differentiation, or make other enhancements. This is certainly evident in the case of relational databases, for which there is a conventionalized set of abstractions (the concepts associated with relational tables, indexes, etc.) and also some standards related to access to those abstractions (SQL and enhancements such as ODBC and JDBC to support queries across software interfaces). But it is also the case that the particular vendors may add specialized features to these standard interfaces to support new capabilities in response to the market. In other words, although there is a seemingly inevitable commoditization of software component capabilities, there is also a seemingly indefinite deferment of reaching the goal of fully predictable decision outcomes regarding innovative software-manifest capabilities in systems. This is a key point about the intrinsic lack of limits of software (except those we choose to impose, such as architecture—see Chapter 3), and indeed this is a principal characterizing feature of software as an engineering building material.

Finding 1-5: It is dangerous to conclude that we are reaching a plateau in capability and technology for software producibility. To avoid loss of leadership, the DoD will need to become more fully engaged in the innovative processes related to software producibility.

The Continued Maturing of the Software Discipline Creates New Challenges

A consequence of this phenomenon of continuous capability improvement is the challenge of integrating innovative software development into mainstream systems engineering processes. Despite the pervasiveness of software and its pivotal role in systems and infrastructures, the engineering of software has not yet matured into a fully rigorous discipline. For innovative software projects, fully measurable and predictable process flows and outcomes—a hallmark of conventional process maturity—will necessarily remain elusive.

This is partly due to the fact that each new technical advance in software not only creates opportunities but also presents new difficulties of measurement and risk assessment. Engineering repeatability is achievable only for the more precedented systems. Additionally, a characteristic of many innovative projects is that the scope of the intended impact may be defined while specific details regarding functional and quality attributes emerge only in the course of development. Fully elaborated requirements against which predictions can be made often do not (and in many cases *should not*) exist.[45] In other words, "predictability" may have more to do with success in addressing a need and less to do with how that need is specifically addressed.

Of course, this is true of nearly all other engineering disciplines. A key difference with software is that development of particular software functionalities, once routinized, is then quickly automated. The result is that expensive custom development gives way to much-lower-cost component procurement. In turn this often gives way to open-source availability of the same functionality. Moreover, unlike other areas of engineering, the intrinsic cost of replicating or deploying software artifacts is near zero. We can conclude that the effect is that a relatively much larger portion of the overall engineering effort in a software enterprise is devoted to creating specifically innovative functionalities. In other words, once development of specific software functionalities is routinized, the cost can vanish relatively quickly, which means more of the overall hands-on engineering activity is in the realm of the innovative and unprecedented.

Additionally, the risks and difficulties of software are growing in severity and diversity, and we continue to experience failures of all kinds—related to reliability, security, flexibility, and other attributes. Software-related problems are responsible for life-threatening failures in health devices, failures of space missions, failures in military systems, cascading failures in infrastructure for telecommunications and power utilities, and so on.

This may create a perception that there is an unavoidable trade-off between precedent and project risk, and that the only way to avoid major project completion risk is to compromise on ambitions regarding system capability. The committee's analysis suggests that this is not necessarily the case, and in Chapter 2, the committee considers the means by which the engineering risks associated with innovative projects can be mitigated incrementally, thus potentially reducing the cost and project completion risk without overly compromising functionality.

Conclusions Regarding Software Innovation

The committee draws several conclusions from these observations regarding software capability improvement. First, mere presence in the market as a software user requires keeping pace with ongoing software innovation and improvements to practices. This is true even for individual software components—indeed, commercial software managers recognize that software starts to "die" (in the sense of becoming less valuable to users) the moment it stops evolving. It is also true for practices—continuous improvement in practices and processes is essential for survival. Desktop computers now are almost

[45] See, for example, the description of the double helix design methodology in BG Harry Greene, USA, Larry Stotts, Ryan Paterson, and Janet Greenberg, January 2010, "Command Post of the Future: Successful Transition of a Science and Technology Initiative to a Program of Record," *Defense AR Journal,* Defense Acquisition University. Available online at http://www.dau.mil/pubscats/PubsCats/AR%20Journal/arj53/Greene53.pdf. Last accessed August 20, 2010.

1 million times more powerful than those of 1980. The software capabilities have similar jumps, although these jumps are less apparent and not easy to measure.

Second, leadership in the market, as a producer or consumer, requires an active organizational role in defining the architecture of systems, and doing so as a first mover or fast follower. Software economics are focused on externalities—where the technical manifestation of the system structure is the software architecture and internal framework and component interfaces. This requires sustained technological leadership and clear thinking about the significance of architectural control. This is particularly significant in the definition and leadership of the design of the major ecosystems. Some of these are wholly controlled by commercial vendors, but others involve complex community processes.

Third, software technical challenges are broadening. These include, for example, software assurance, ultra-scale architecture, concurrency (multi-core and distributed), framework design, programming language improvements for assurance and scale, concepts for "big data" systems, and so on. These challenges are addressed in Chapters 2, 3, and 4 of this report.

Fourth, risk management models need to be continually adjusted to accommodate the new realities of software and of IT-enabled business practices, as noted above. This is the subject primarily of Chapters 2 and 4.

Finally, the role of software leadership in the global economy is growing, and this is increasingly recognized, with the result that global competition is becoming more intense at every level of capability. Overseas competition is greatly facilitated by the low barrier to entry—costly physical facilities are not needed in the software economy, but education and technical currency are fundamental and ongoing challenges. This is significant for the DoD, which has counted on U.S. industry, including defense contractors and their supply-chain participants, to sustain technological leadership in software as a key driver of capability leadership in systems. Such leadership must be maintained through constant investment in innovation and in people. At the highest level of technical sophistication, this requires investment in university research to produce a sufficient pipeline of technical leaders.[46]

[46] U.S. PhD students in computer science and IT-related areas are almost universally supported with tuition and stipends covered by sponsored research. Universities rarely have funds to provide direct fellowships to PhD students in these areas, and few students have the resources or capacity to self-fund or to take on (often additional) loans to cover their costs.

2

Accept Uncertainty: Attack Risks and Exploit Opportunities

INNOVATION, PRECEDENT, AND DYNAMISM

Experience shows that there is a correlation between the degree of software system precedent, routinization, and stability, on the one hand, and the ability to deliver results with predictable cost, schedule, and operational test and evaluation (OT&E) success, on the other. Many of the Department of Defenses's (DoD's) information technology (IT) systems are precedented, as are significant portions of mission systems. These include the office automation and back-office systems for business operations that are increasingly conventionalized in both commercial and national security contexts. There are precedents for such systems in numerous institutions and environments. Such conventions enable the DoD to build on wide internal experience, other government experience, and commercial experience, reducing the uncertainty associated with predicting the outcomes of particular design decisions. This happens when similar decision points have been experienced in other settings, experience was gained, and it has been possible to transfer that experience to new projects that are sufficiently similar. For precedented development efforts, managers can project plans further into the future of a development process with higher accuracy. They can focus more closely on optimizing cost, schedule, and other factors while managing the various tradeoffs involved. For these routine systems, the DoD benefits when it can adjust its practices to conform to government and industry conventions, because it is then able to build more directly on precedent and also exploit a broader array of more mature market offerings.

The largest producibility challenges for the DoD come from its need for unprecedented, innovative systems that can be rapidly adapted. The mission of the DoD requires it to constantly move forward in advancing the capability of its systems. The committee uses the term "unprecedented" to refer to systems concepts, designs, or capabilities that are not similar enough to the existing base of experience to benefit from fully following an established pattern. As a result, development efforts may involve greater risk (see next section). This report calls these innovative and agile projects *software-intensive innovative development and reengineering/evolution* (SIDRE) efforts and focuses much of its attention on them. It must be recognized, however, that most unprecedented systems designs, including very-large-scale interlinked systems, generally incorporate significant portions that are themselves precedented and possibly also associated with established commercial or open-source ecosystems.

Precedented Systems

Put simply, engineering practices and technology choices for precedented systems (such as stable back-office systems capabilities) are guided by convention (such as commercial best practices for such systems), while engineering practices and technology choices for unprecedented systems and components are guided by processes for mitigating engineering risk. In fact, there is a constant pace of innovation even for seemingly established functional capabilities, such as back-office systems, with some areas of innovation and other areas that are more guided by convention.

Following precedent may require engaging in a process of adapting previously unique business practices to reflect more "standard" or "conventional" operations practices that are more readily supported within mainstream systems and ecosystems, particularly when there is correspondence with modern back-office systems in the commercial world. This adaptation of functional goals to achieve consistency with normative practice is an explicit part of the commercial requirements engineering process. The DoD and other government agencies may struggle more because they may find it more difficult to compromise, and for many good reasons. But the extent to which the DoD can find commonalities (and avoid unnecessary differentiation) with other government agencies creates opportunities for major cost reduction, risk reduction, and process simplification.

Unprecedented Systems

As noted in the previous chapter, the need to develop unprecedented systems is a consequence of the highly complex and rapidly evolving operational environment within which the DoD must operate and execute its mission. Complexity is increasing, as is the difficulty of the threats and challenges. Highly capable information technology is now ubiquitous worldwide, and adversaries have ready access to cutting-edge technology. Mission and deployment priorities are constantly shifting. The DoD must collaborate extensively with other agencies, nongovernmental organizations (NGOs), coalition partners, and others in constantly changing configurations over which the DoD has no control. Operational decisions are derived from a broad diversity of inputs. Command-and-control models must adapt to rapidly evolving threats. Success in this environment depends on systems designed for flexibility, agility, and robustness, but it also requires flexibility, agility, and robustness in the process by which systems are developed and continue to evolve. There is much less opportunity to rely on precedent and much greater requirement to undertake a process of ongoing innovation. This process of innovation entails acceptance of certain categories of risks. (See Box 2.1 for details.)

Commercial best practices have also evolved for developing unprecedented systems. Air traffic control, telecommunications switches, middleware (such as from IBM and Oracle), operating systems (such as from Apple and Microsoft), and large-scale web applications (such as from Google, Facebook, and Amazon) have been developed under commercial best practices with varying degrees of success. However, these large-scale, unprecedented systems emerged over a period of years from market opportunity without a specification-driven need, while others did not. Besides business savvy, the main critical success/failure factors in these situations have involved the ability to assess potentially disruptive technologies and competitor strengths, and the corporate agility to adapt to change.[1]

This chapter addresses the processes and practices by which these risks can be understood and addressed in the engineering of systems. A principal conclusion is that a well-managed incremental (iterative) process, supported by appropriate evaluation and measurement approaches, can more reliably lead to successful outcomes even when there are significant engineering risks. On the other hand, attempts to produce innovative or unprecedented systems using familiar linear ("waterfall") processes

[1] Michael Cusumano and David B. Yoffie, 1998, *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft*, New York: The Free Press. See also Clayton M. Christensen, 1997, *The Innovator's Dilemma: The Revolutionary Book That Will Change the Way You Do Business*, New York: Harper Business. See also Robert L. Glass and P. Edward Presson, 2001, *ComputingFailure.com: War Stories from the Electronic Revolution*, Upper Saddle River, NJ: Prentice Hall PTR.

can very often lead to unhappy surprises—late-breaking negative feedback regarding early design commitments that, when learned at a late stage in the process, can be very costly to revise. That is, what appears to be a "safe," conservative decision to follow the most basic process is in fact a dangerous decision that can drastically increase programmatic risk—and the possibility of total project failure.[2] The key features of a well-managed incremental process for innovative systems are (1) measurements that are informative and relevant, and (2) process feedback loops that are relatively short, with potential major reductions in programmatic risk.

For many of the innovative systems at the heart of the DoD's software producibility challenge, the details of the future requirements are not—and in many cases cannot be—fully understood. Thus the need to innovate is made more challenging by the simultaneous need to be agile as requirements necessarily evolve over time.

MANAGING RISK AT SCALE

There are attempts to manage innovative software development following process patterns more appropriate to precedented systems and to established predictable engineering disciplines. One consequence is that linear development processes are inappropriately used despite the presence of high engineering risk (and requirements risk also), with the consequence that those engineering risks are unnecessarily transformed into increasing project risks. A second consequence is that there is unjustified emphasis on achieving excessive precision at the outset regarding functionality desired by the user, choices of infrastructure platforms, and possibly also economic tradeoffs in various complex dimensions of quality. This drive for excessive precision in these areas can yield a surfeit of specifications and other early design artifacts, which may in fact give only false comfort—and lead to downstream scrap and rework.

This is because these process patterns do not account for the engineering risks and uncertainties inherent in developing innovative software, where there are no laws of physics and materials to constrain solutions to particular structural patterns. In precedented software, the structural patterns derive from established software ecosystems and from the body of precedent. In innovative SIDRE systems, these patterns are lacking, which is both advantageous, in that opportunity is afforded for innovation and creativity, and also disadvantageous, in that greater levels of uncertainty must be addressed.

Modern governance approaches for larger systems must account for the management of uncertainty. At scale, they must exploit collaboration among distributed teams and in rich supply chains for which there is a continuous negotiation of scope, quality, and resources to balance the opportunities in delivering more value with the uncertainties inherent in software development cost and scope targets. That is:

It is important to treat scope, plans and resources as variables (not frozen baselines) and explicitly manage the variances in these variables until they converge on acceptable levels to commit a project/product to full scale production.

Fortunately, recent DoD and NRC studies[3] have resulted in some very initial steps, as evidenced in

[2] Some program managers sarcastically refer to an inappropriately used linear (waterfall) process model as the "requirements, delay, surprise" process model. Fred Brooks's recent book, *The Design of Design* (Boston: Addison-Wesley, 2010), succinctly concludes, "The Waterfall Model is wrong and harmful; we must outgrow it." This point was also made in Fred P. Brooks, 1987, "No Silver Bullet—Essence and Accidents of Software Engineering," *Information Processing* 20(4):10-19.

[3] Assessment Panel of the Defense Acquisition Performance Assessment Project, 2006, *Defense Acquisition Performance Assessment*; see also NRC, Richard W. Pew and Anne S. Mavor, eds., 2007, *Human-System Integration in the System Development Process: A New Look*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11893. Last accessed August 20, 2010; and National Research Council (NRC), 2008, *Pre-Milestone A and Early-Phase Systems Engineering: A Retrospective Review and Benefits for Future Air Force Acquisition*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12065. Last accessed August 20, 2010.

BOX 2.1
Programmatic, Engineering, and Systems Risk

Programmatic Risk

Programmatic or project risks pertain to the successful completion of engineering projects with respect to expectations and priorities for cost, schedule, capability, quality, and other attributes. A principal influence on programmatic risk is the process by which engineering risks are identified and addressed. This applies particularly to engineering risks related to architecture and ecosystems choices, quality attributes, and overall resourcing. With innovative projects, programmatic risk can be reduced through use of iteration, incremental engineering, and modeling and simulation (as used in many engineering disciplines). Programmatic risks that derive from overly aggressive functional or quality requirements, where engineering risks are not readily mitigated, are often best addressed through moderation on the "value side," for example, through scoping of functional requirements. Indeed, for ambitious and innovative programs—those characterized as "high risk, high reward"—for identifying and sorting engineering risks, it is often most effective to focus as early as possible on architecture. Once overall scope of functionality is defined, architecture risks may often dominate the detailed development of functional requirements.

A well-known example of negative consequences of unmitigated programmatic risks is the FBI Virtual Case File (VCF) project.[1] The project is documented in the IEEE Spectrum: "The VCF was supposed to automate the FBI's paper-based work environment, allow agents and intelligence analysts to share vital investigative information, and replace the obsolete Automated Case Support (ACS) system. Instead, the FBI claims, the VCF's contractor, Science Applications International Corp. (SAIC), in San Diego, delivered 700,000 lines of code so bug-ridden and functionally off target that this past April [2005], the bureau had to scrap the US $170 million project, including $105 million worth of unusable code. However, various government and independent reports show that the FBI—lacking IT management and technical expertise—shares the blame for the project's failure."[2]

Eight factors that contributed to the VCF's failure were noted in a 2005 Department of Justice audit. These included "poorly defined and slowly evolving design requirements; overly ambitious schedules; and the lack of a plan to guide hardware purchases, network deployments, and software development for the bureau. . . ." Finally, "Detailed interviews with people directly involved with the VCF paint a picture of an enterprise IT project that fell into the most basic traps of software development, from poor planning to bad communication." (Today, 5 years later, the program has been scrapped yet again.)

Supply chain risk is an area of engineering risk that is growing in significance and that often develops into programmatic risk. This is evident in the DoD's increasingly complex and dynamic supply-chain structure, with particular emphasis on concerns related to assurance, security, and evolution of components and systems infrastructure. This risk can be mitigated through techniques outlined in Chapters 3 and 4 related to architecture design, improved assurance and direct evaluation techniques, multi-sourcing, provenance assessment, and tracking and auditing of sourcing information. Supply chain risk is particularly challenging for infrastructure software and hardware, because of the astonishingly rapid evolution of computing technologies, with commercial replacement cycles typically every 3 to 5 years. In the absence of careful planning, this means that early ecosystem commitments can potentially create programmatic risks in downstream

[1] Ben Bain, 2009, "FBI Pushes Back Completion Date for Sentinel File System " November 10, 2009, *Federal Computer Week*. Available online at http://fcw.com/Articles/2009/11/10/FBI-Sentinel-IG-report.aspx. Last accessed August 20, 2010.

[2] Harry Goldstein, 2005, "Who Killed the Virtual Case File?" *IEEE Spectrum* 42(9):24-35. Available online at http://spectrum.ieee.org/computing/software/who-killed-the-virtual-case-file. Last accessed August 20, 2010. See also James C. McGroddy and Herbert S. Lin , eds., 2004, *A Review of the FBI's Trilogy Information Technology Modernization Program*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=10991. Last accessed August 20, 2010. See also the subsequent NRC letter report, 2004, *Letter Report to the FBI*, James C. McGroddy and Herbert S. Lin, eds., Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11027. Last accessed August 20, 2010.

system-refresh cycles, when shifts in vendor strategy may create unanticipated incompatibilities. Indeed, vendors may often institute incompatible changes in interface specifications in their ecosystems in order to force user organizations to stay current with evolving technology. Thus, the advantage of "riding the curves" with off-the-shelf infrastructure must be weighed against the loss of control by program managers. But this is not a simple tradeoff, since savvy architects have ways to structure systems to increase the possibility of "having it both ways" in many cases.

Related to supply chain is another kind of programmatic risk, derived from conflicting business incentives. This kind of risk is often present in projects focused on enhancing interoperation among systems. With interoperation, for example, data from sensors for one system can be used to enhance situation awareness models in another system. Indeed, one of the strong arguments for net-centric approaches is the benefit of broad sharing of sensor data to enhance situation awareness and better inform tactical decision making.[3] Despite the natural drivers for interlinking, there are risks and difficulties. A critical system risk, for example, relates to security—poor architectural decision making at the outset could mean that successful attacks could result in amplified consequences, due to the larger scale of the overall system.

There are also programmatic risks relating to potential conflicts in the business and mission interests of the organizations responsible for the entities being interlinked. Vendors, for example, who are competing at the level of ecosystems may see interlinking as an opportunity for competitors to benefit from network effects associated with acceptance of the ecosystem by users—that is, interlinking with competitor systems may be perceived as a threat to investment in ongoing ecosystems enhancement. There are also circumstances under which DoD contractors may also see enhanced interoperation (and open architectures, more generally) as a threat to lock-in and an enhancement to opportunities of competitors.

Engineering Risk

Engineering risks pertain to uncertainties and consequences of particular choices to be made within an engineering process. High engineering risk means that outcomes of immediate project commitments are difficult to predict and consequently raise programmatic risk. Engineering risks can relate to many different kinds of decisions—most significantly architecture, quality attributes, functional characteristics, infrastructure choices, and the like. Except in the most routinized cases, much of the practice and technology of software engineering is focused not only on system capability, team productivity, and resource requirements for development, but also on the reduction of the engineering risks that unavoidably arise in unprecedented developments.

As DoD and commercial systems evolve with more and more functionality delivered in software, systems engineering and software engineering techniques are intersecting, leading to a critical area for new research and the advancement of practice. The challenge is three-fold. First, traditional decision support techniques need to be enhanced to address the diverse kinds of software engineering risks. Second, there is need for modeling, simulation, prototyping, and other "early validation" techniques for many different kinds of software engineering decisions, for example, those related to architecture, requirements, ecosystem choice, tooling and language choice, and many others. Third, system engineering models must be developed through which appropriate "credit" can be given in earned value models for activities that mitigate identified engineering risks. This entails addressing a range of challenges related to measurement and process design. Development of techniques to meet these challenges would benefit commercial industry as well as the DoD.

An example of the identification and resolution of engineering risk is described in a workshop report

[3] Metcalfe's Law is an observation on network effects, stating that the "value" of a telecommunications network grows with the square of the number of nodes in the network—when the number of nodes in a complete graph doubles, the number of edges roughly quadruples. Of course, there are other ways scale influences "value" that may make actual value greater or less than quadratic. But regardless of the model, it is clear that the effects are super-linear. This observation explains the forces that drive the coalescing of separate networked systems into aggregates, including the internetworking initiatives of the 1970s that coalesced diverse computer networks into the Internet.

BOX 2.1 Continued

produced by this study committee.[4] This case study concerns the internal ecosystem at Amazon.com. Amazon evolved from a relatively straightforward e-commerce model into a highly complex aggregation of sellers and buyers, with a business model that depended in part on realizing the synergies of this aggregation and the growth in scale. "Amazon builds almost all of its own software because the commercial and open source infrastructure available now does not suit Amazon.com's needs." When it became clear that initial architectural approaches needed to be enhanced, a critical decision point was reached, with developers at Amazon facing a choice between a "goal of building the 'perfect' system (the 'right' system) whatever the cost" and a very different and "more modest goal of building a smaller, less ambitious system that works well and can evolve." Developers recognized that trying to build the best possible system was a long, difficult, and potentially error-prone process. It also forced anticipation of a comprehensive set of potential downstream business models. The developers instead adopted an approach designed to support evolution and rapid organic growth. It necessarily embodied fewer assumptions regarding the business model, but it was designed to be adaptive and robust. This led to greater emphasis on infrastructure performance, scalability, and reliability, with a focus on implementation ideas such as redundancy, feedback, modularity, and loose coupling, under rubrics such as "purging," "spatial compartmentalization," and "apoptosis." This was the model that led to Amazon's rapidly growing venture into cloud computing and associated services.

Systems Risk

Systems risks pertain to the potential hazards—operational risks, mission risks, deployment challenges, and so on—associated with the deployment of a system. What are the kinds of failures, and what kinds of hazards do they create? For example, cascading failures have been experienced in telecommunications and power utilities. These are large-scale system failures resulting from unwanted positive feedback of local failures triggering failures elsewhere, leading to more global failures with the corresponding hazards. That is, the hazard of a single system failing can often be associated with a much larger aggregate of systems, often spread across a wide geography. The consequences of a single local failure thus extend well beyond the immediate locality of the failure. The hazard is at a much greater scale.

A case study in systems risk is the Toyota Prius, a "highly computerized car,"[5] that relies on software programs to manage the various applications and features of the vehicle. This complex system is really an amalgam of simpler subsystems interoperating with each other across a network fabric. Drivers of the 2010 Prius had reported brake malfunctions, which would later be attributed to a glitch in the software controlling the car's brakes. It is unclear from reports from Toyota and in the press whether the "software glitch" was an algorithmic fault faithfully encoded into the software or a fault in the software encoding or the software infrastructure. Regardless, the repair of the fault was accomplished through software updates: Toyota later issued a software patch for the brake problem.[6] In February 2010, Ford also resolved a braking issue through a software upgrade.[7]

---

[4] NRC, 2006, *Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 20, 2010.

[5] Stephen Manning and Tom Krisher, 2010, "More Trouble for Toyota as Regulators Launch Investigation of Prius Brake Problems," *Associated Press*, February 4, 2010. Available online at http://autos.ca.msn.com/news/canadian-press-automotive-news/article.aspx?cp-documentid=23387474. Last accessed August 20, 2010.

[6] David Millward, 2010, "Toyota Offers UK Prius Owners Brake Software Upgrade," *Telegraph.co.uk*, February 8, 2010. Available online at http://www.telegraph.co.uk/motoring/news/7189917/Toyota-offers-UK-Prius-owners-brake-software-upgrade.html. Last accessed August 20, 2010.

[7] David Bailey, 2010, "Ford offers fix for Fusion hybrid brake glitch," *Reuters.com*, February 4, 2010. Available online at http://www.reuters.com/article/idUSTRE61369I20100205. Last accessed August 20, 2010.

the DoD's revision of DoD Instruction 5000.02 and the recent Congressional Weapon System Acquisition Reform Act that establish such convergence as DoD acquisition policy.[4] However, the policy does not provide detail about how such convergence can be achieved, particularly in the software arena.

Finding 2-1: Modern practice for innovative software systems at all levels of scale is geared toward incremental identification and mitigation of engineering uncertainties, including requirements uncertainties. For defense software, the challenge is doing so at a larger scale, and in ways that are closely linked with an overall systems engineering process.

Innovation and agility are related. Innovation is the ability to create new systems concepts to address emerging challenges and opportunities. Innovation can be in concept, functionality, architecture and design, performance, and so on. Innovative functionalities have migrated into software realizations because of the special characteristics of software. By improving our capability to manage uncertainty, we are able to accelerate the delivery of more capable systems and reduce costs. The environments of defense mission needs and of computing technology are both rapidly changing and often in unpredictable ways. This creates uncertainty, particularly when systems must be designed to anticipate these changes in mission and technology over periods of many years. Not all changes can be anticipated, which implies that not only must architecture and design be forward-looking, but also that ongoing process must be agile in facilitating ongoing innovation in response to changing needs and opportunities. What does it mean to "manage uncertainty," and what are good characterizations and, where possible, measurements of the various dimensions of uncertainty?

It is often stated as a matter of principle that we must measure something if we are to manage it. However, in the history of software engineering, the principal "measurables" have been time, effort, lines of code produced, and defects found and fixed. These are only approximate surrogates for the attributes of progress that matter in complex development projects, such as identification and resolution of engineering risks, assurance with respect to quality attributes, manifestation of critical functional features, ability to support future evolution, and so on. The former set of measurables (e.g., time, effort, etc.) are perhaps more useful for linear or waterfall developments, but they are of diminishing value for innovative and agile projects. In these projects, not only must engineering risks be identified and resolved, but also observable attributes must be created to provide evidence—and reduce the possibility of "going into denial" regarding challenging engineering risks. This issue is elaborated in the section below on earned value concepts.

From the perspective of quantitative measurement, the uncertainty of software producibility can be understood through the following observations:

1. Best practices for software development resource estimates employ empirical, parametric models. These models typically have 20 to 30 input parameters and produce a probability distribution of outcomes.
2. The variance of the distribution of outcomes is a good measure of the uncertainty.
3. Managing this uncertainty means reducing the variance in the distribution of outcomes as estimates to complete are recalculated on a periodic basis.

Process agility is needed to respond rapidly and effectively to changing circumstances. But often those changing circumstances are in the form of late-breaking (in a development process) understanding of key design commitments and emerging engineering risks. Thus effectiveness at the management

[4] The 2010 National Defense Authorization Act (NDAA) language (Section 804) is also evidence of progress. National Defense Authorization Act (NDAA) of Fiscal Year 2010, Pub. L. no. 111-84, 111 Congress, (2009). Available online at http://www.wifcon.com/dodauth10/dod10_804.htm. Last accessed August 20, 2010.

of uncertainty is one key to process agility in systems definition and conceptualization, as well as realization.

Technology cannot overcome poor governance or management. However, in the past two decades, considerable progress has been made in assessing organizational capability to enact process models to manage complex software developments. This understanding has been packaged in a variety of ways, such as the capability maturity model, ISO 9000, and the spiral development model. Even well-managed software development can be hamstrung by poor selection of technology, but it can also be enhanced significantly by the judicious use of technology. Software-intensive systems are complex. Tools are needed to help manage complexity, track changes, maintain configuration control, and enforce the integrity of the architecture—these help teams avoid mistakes often driven by that complexity. Indeed, modern software teams at all levels of scale rely much more intensively on tooling for process support than at any point in the past.

Software is distinctive in that it permits rapid iterations, aggressive prototyping, simulation, and modeling, along with other techniques that can afford early validation with respect to many critical acceptance criteria. Improved software infrastructure and practices also enable agility, as do improved means to measure and assess software quality and other attributes. The governance and management process for unprecedented systems can better exploit these unique software capabilities. This means, in particular, the aggressive use of iterative risk-managed processes and the definition of suitable earned value measures related to validation of requirements and architecture, team collaboration, and continuous integration. It also means that platform automation support for measurement, resource estimation, variance reduction, and change propagation must mature. Another recent study from the National Research Council has assessed the potential, primarily from a management perspective, for the DoD to more widely employ incremental and iterative processes to support risk-managed development of SIDRE systems.[5] The recommendations of this study are generally in harmony with the recommendations of this report, which focuses more on technological enablers and on attendant research and technology-development challenges.

Earned Value Management and Unprecedented Systems

Earned value management (EVM) is "a means of determining the financial health of a project by measuring whether the work completed to date is in line with the budget and schedule planning." One of the goals of using EVM is to get early warning of potential problems. EVM tracks planning, progress, cost, earned value (the planned cost of actual progress), and variance in cost and schedule.

Although the technique is seemingly straightforward, the application of EVM for innovative and unprecedented software-intensive systems poses challenges. In particular, assessing and measuring actual progress is difficult. Conventional EVM systems make several assumptions, namely: (1) The relationship between resources and progress is linear, (2) The effort needed to meet certain goals is predictable at the outset, (3) Progress is easily and accurately measurable, and (4) The expected outcome—as articulated in requirements—is well understood. None of these assumptions applies in the case of software-intensive unprecedented system development efforts where the level of uncertainty changes the governance process from planning and tracking a straightforward production sequence of related tasks to an emerging discovery process that requires continuous steering.

Extending EVM to SIDRE software requires some significant changes in how EVM assessment and measurement strategies are applied. In particular, EVM in this context needs to be adapted from tracking conformance to planned expenditures to steering toward planned value creation. For this to happen, significant improvements are needed in our ability to value software assets. For example, a major, unfinished software asset is no more than an option to guide further investments that, with some

[5] NRC, 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, D.C: National Academies Press. http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 20, 2010.

remaining risk, will lead to a finished product that creates realized value for an organization. In other words, software systems present very weak observables.[6]

But, such a product contains additional value in the form of flexibility to be better adapted, through additional investments, to its evolving operating conditions. Placing value on adaptation flexibility is essential for reasoning about investments in modular design architectures, such as those produced by the application of product-line approaches. Unfortunately, except at the level of overall ecosystems and vendor components, the value of most software design assets may be apparent only within a project and may change according to architectural choices. Thus, valuation of these assets is difficult and risky. We have neither the models we need to perform such valuation, nor adequate approaches to develop and validate estimates needed as inputs to such models (e.g., of uncertainties about future conditions).

There is perhaps the potential to calibrate models over time based on past experience, though calibrations are always vulnerable to invalidation as operating conditions change. Nevertheless, some kind of approach to valuation (not only accounting for costs but also for value created, even if in the form of options) is important to managing iterative or other development processes to optimize for value created, rather than merely for conformance to predicted cost flow streams.

Time-Certain Development and Feature Prioritization

The fact that (particularly SIDRE) software development effort and duration cannot be estimated precisely means that it is unwise to try to lock a software project into simultaneously fixed budget, schedule, and feature content (as has been found in many fixed-price, fixed-requirements software development contracts). The concept of time-certain development recommended in the Defense Acquisition Performance Assessment (DAPA) report and elsewhere[7] avoids this problem by fixing duration as the independent variable and feature content a dependent variable. This is basically the same concept as the agile practice of *timeboxing*, but it needs more success conditions for large, mission-critical projects for which, as time is running out, it is difficult (and time-consuming) to determine which features to drop, and how to drop them without adverse side effects.

The critical success conditions for large-project time-certain development are to prioritize the features in advance and to modularize the architecture to make it relatively easy to add or drop borderline-priority features. In evolutionary development, this does not mean that the features will never be available, but that they will be deferred to a later increment. Prioritizing features in multi-stakeholder situations is never easy, but it becomes easier if the decision is just to determine what features are most needed in the next increment.

A most significant side effect of feature prioritization is that it produces a consensus ranking of the relative value of the system's features. This provides the beginning of a way to reason about project risk, as the key quantity in risk management is an item's risk exposure, defined as the product of the probability of loss times the size of the loss, which is known at least relatively from the feature prioritization.

[6] This is in the sense of the traditional OODA (observe, orient, decide, act) loop, which underlies iterative processes and incremental development. That is, as iterations and increments of effort yield results, future iterations and increments necessarily build on those results. The challenges of software measurement and evaluation (as addressed throughout this report) relate to the "observe" part of the loop, the process-related challenges relate to the "orient" and "decide" parts of the loop, and many of the architecture/design and programming challenges relate to the "act" part of the loop.

[7] This includes, most significantly, the NRC report on *Achieving Effective Acquisition of Information Technology in the Department of Defense*. NRC, 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, D.C: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 10, 2010. Last accessed August 20, 2010.

Evidence-Based Software Engineering and Risk Probability

Another major trend in software engineering and project management is to shift from schedule-based milestones ("The contract says that the Preliminary Design Review (PDR) will be held on April 1, so that's when we'll hold the PDR, whether we have a design or not.") to event-based milestones ("We'll finish all the design features by June 1, so we'll hold the PDR then."). However, such reviews often fail because there is no way to tell from all the Unified Modeling Language (UML) diagrams and PowerPoint charts whether the design will scale-up, handle crisis conditions, meet critical timelines, or be buildable within the available budget and schedule.

This has led to the current trend toward evidence-based milestones, and evidence-based software and systems engineering in general. This approach places responsibility on developers not only to create artifacts for review such as operational concepts, requirements, designs, plans, budgets, and schedules, but also to produce evidence that if a software system were built to the design, it would satisfy the requirements, support the operational concept, and be buildable within the budgets and schedules in the plan. This evidence would then be reviewed by independent experts, and shortfalls in evidence would be treated as uncertainties or probabilities of loss. As with the relative sizes of loss determined from requirements prioritization above, these probabilities are generally known only relatively, but they can be combined with the relative sizes of loss to produce at least relative risk exposure quantities for use in risk management.

Actually, evidence-based software and systems engineering has been practiced many times and has been a consistently performed and high-payoff corporate practice at leading companies such as AT&T since the 1980s.[8] However, such evidence is usually asked for in contract data item descriptions (DIDs) in optional appendices, where it is one of the first things to go when resources become strained. Making appropriate evidence a first-class deliverable not only would ensure its development, but also would make it an element of earned value management, in that it thus would have to be planned for and its progress tracked with respect to the plans. The evidence should be parsimonious and focus on enabling of action—rather than on the massing of "read-never" program documentation. These points are summarized in the following findings and recommendations.

Finding 2-2: The prescription in DoD Instruction 5000.02 for the use of evolutionary development needs to be supplemented by the development of related guidance on the use of such practices as time-certain development, requirements prioritization, evidence-based milestones, and risk management.

Finding 2-3: Extensions to earned value management models to include evidence of feasibility and to accommodate practices such as time-certain development are necessary conditions to enable successful application of incremental development practices for innovative systems.

As noted throughout this report, the DoD would benefit from investing effort in developing improved quantitative measures related to diverse software attributes such as quality, productivity, architecture compliance, architecture modularity, process performance, and many others. But DoD practices must also recognize that existing metrics do not fully reveal critical attributes of systems and process status and that expert judgment also has a critical role, particularly with respect to architecture, design, and many quality attributes associated with SIDRE systems. Evidence-based software and systems engineering approaches are being increasingly applied to address achievement of critical SIDRE attributes and need to be better institutionalized into DoD acquisition practice.

[8] Joseph F. Maranzano, Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss, 2005, "Architecture Reviews: Practice and Experience," *IEEE Software* 22(2):34-43.

Finding 2-4: Research related to process, measurement, architecture, and assurance can contribute to the improvement of measurement practice in support of both routine management of engineering risks and value assessment as part of earned value management.

For example, keys to developing cost-effective evidence involve determination of feature and attribute priorities, assessment of candidate evidence-generation capabilities (modeling, simulation, prototyping, bench marking, exercises, early working versions, citations of relevant previous experience), and measurement of progress toward thorough evidence generation.[9] Some initial steps in this direction are provided in a report by Boehm and Lane.[10]

Recommendation 2-1: The DoD should take aggressive actions to identify and remove barriers to the broader adoption of incremental development methods, including iterative approaches, staged acquisition, evidence-based systems and software engineering, and related methods that involve explicit acknowledgment and mitigation of engineering risk.

There are different kinds of barriers that can be addressed through combinations of established best practice and emergent improved practice derived from technology and other improvements. These potentially surmountable barriers include (1) improved measurement and associated technology, (2) architecture validation using models, simulation, prototyping, etc., (3) program manager training and evaluation of perceived career risks (see findings below), (4) accretion of an accessible experience base and other shared resources that can facilitate sound decision making, and (5) acceptable shifts of early-stage emphasis for innovative systems from detailed functional requirements to concurrent engineering of requirements, architecture, process definition, and evidence of their compatibility and feasibility. Similar barriers exist in commercial industry, of course. These are accentuated in DoD because of its particular challenges of arm's-length contractual relationships, high assurance requirements, potential presence of adversaries in the systems development activity, and other barriers.

MANAGING REQUIREMENTS AND ARCHITECTURE

Software development complexities tend to increase non-linearly as systems scale up in complexity, features, and quality goals. The challenge for the DoD is that its requirements must be addressed at unusual scale, complexity, interconnection, security, and with life-critical mission requirements. This challenge is exacerbated by the fact that the DoD is not sufficiently exploiting known techniques for the management of complex and evolving requirements. These techniques have been a focus of research for many years, but the known techniques are not widely employed on DoD applications—techniques including spiral development, joint application development, agile development, etc. The resulting difficulties are well known.[11,12]

There is widespread agreement that the requirements-delay-surprise (linear) approach to software development is not effective for innovative systems. The committee proposes more extensive use of an incremental, risk-assessment-driven approach. It is important to appreciate that, for incremental approaches to succeed, there needs to be forward-looking up-front investment in the overall system and process design. This enables problems to be decomposed in such a way that engineering risks can be identified, initial architecture models developed, and overall programmatic risk is minimized. If this is

[9] This concept of evidence generation is different but analogous to the discussion of evidence-based assurance in Chapter 4.

[10] Barry Boehm and Jo Ann Lane, 2010, *Evidence-Based Software Processes*, Proceedings, *2010 International Software Process Conference* Springer, Berlin.

[11] NRC, 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, Washington, DC: National Academies Press, Washington, DC. Available online at http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 20, 2010.

[12] Barry Boehm and Richard Turner, 2003, *Balancing Agility and Discipline: A Guide for the Perplexed,* Boston: Addison-Wesley.

done well, then the subsystems can be developed relatively independently.[13] As more information about the constraints and limitations on the subsystems becomes clearer, this high-level system architecture—the minimum set of critical common commitments—needs to be repeatedly revisited, reevaluated from a technical perspective, and evolved to assure that it remains appropriate. This reevaluation should be supported by a range of modeling and analysis capabilities to help detect problems and vulnerabilities at the earliest possible stages.[14] Similar evaluation should be applied to the overall system architecture, as well as to each of the subsystems.

Changeability and Correspondence

Requirements need to be seen as an evolving set of goals and constraints. Initially, the requirements capture the scope of the mission and associated operating environment and goals of the system. As design decisions are made, scope is refined and these high-level goals need to be mapped down to the lower-level details.[15] Design-specific constraints need to be included, and newly realized goals need to be added. Requirements are not useful, however, unless they are utilized throughout the development, maintenance, and evolution process. Requirements developed early in the project need not only to continue to be used to drive design decisions, but also to drive the architecture-level validation, low-level design validation, and implementation validation and testing. Recent work on model checking, program analysis, formal verification, and testing (and their interaction) demonstrate how an evolving requirements base can impact upstream and downstream activities. Requirements and associated systems must also adhere to the principle of *correspondence*, which states that minor changes in requirements should generally mean only minor system changes. Thus defining "minor" with regard to requirements is critical, and aligning architecture with overall functional and quality attribute requirements is essential. It should be noted, however, that the best architecture is often a discontinuous function of the performance requirements: a common example is the different scalability of different COTS products. Other key needs are for evolution requirements that specify the expected growth in workload that the architecture must support for developer evidence that the architecture will support not only the early increments, but also the full operational capability.

The value of this approach is reinforced when experience is considered in "adjustment" or "renegotiation" of requirements. This experience suggests that, despite the intent of non-negotiability as implied by the use of the word "requirements," we nonetheless see requirements being changed, bartered, and negotiated on almost every successful project. Changing a requirement receives tremendous scrutiny because it usually has an impact on the contract among stakeholders.

Scope

Scope, as distinct from hard requirements, is intended to simply represent the current state of our understanding of the overall operational context and the needs that are addressed. In the committee's experience, successful software projects are managed in a way that, implicitly, treats scope as a variable,

---

[13] This was a goal of the Amazon.com architecture reengineering project, as documented in NRC, 2007, *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 10, 2010.

[14] Multiple studies have explored the relative cost of finding and repairing defects as a function of stage of the process. The cost differential between early and late stage can be two to three orders of magnitude. See e.g. See, RTI, 2002, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Planning Report 02-3, RTI Project Number 7007.011. Available online at http://www.nist.gov/director/planning/upload/report02-3.pdf. Last accessed August 20, 2010.

[15] In some innovative programs, there is a simultaneous refinement of system operational concept and technology design. An example of this is the "double helix model" adopted in the Command Post of the Future program, detailed in BG Harry Greene, USA, Larry Stotts, Ryan Paterson, and Janet Greenberg, January 2010, "Command Post of the Future: Successful Transition of a Science and Technology Initiative to a Program of Record," *Defense AR Journal,* Defense Acquisition University. Available online at http://www.dau.mil/pubscats/PubsCats/AR%20Journal/arj53/Greene53.pdf. Last accessed August 20, 2010.

just as cost and resources, schedule and work breakdown, quality (with respect to various attributes), and other overall constraints are variables subject to ongoing negotiation. An example is the prioritization of requirements and use of scope as a dependent variable in time-certain development. Even when scope is treated as a seriously scrutinized and controlled variable in DoD projects, a collaborative, open, and honest management style between customer and contractor, supported by constant effort to improve measurement and observation capability, has proven to be another necessary ingredient for success.

These are the foundations of modern agile governance that demand executable capability demonstrations over time. Modern agile governance of software delivery means managing uncertainty through steering. In a healthy software project, each successive phase of development produces an increased level of understanding in the evolving plans, specifications, and completed solution, because each phase furthers a sequence of executable capabilities and the team's knowledge of competing objectives. At any point in the life cycle, the precision of the subordinate artifacts should be in balance with the evolving precision in understanding, at compatible levels of detail and reasonably traceable to each other.

ESTIMATIONS, CONTRACTING, AND ITERATIVE DEVELOPMENT

The DoD operates within a complicated federal procurement and acquisition process. From the outset the government typically awards to the contractor with a viable technical solution and the lowest cost. For software-intensive systems there is a conventional wisdom that aggressive bids have driven many programs to diminished probabilities of success.[16] Although the reasons for cost overruns and delays are complex, the choice of evaluation criteria in this process is undoubtedly a factor. The government and contractors need to establish rigorous processes to ensure that we have a basis for size estimates that have sound derivation from comparable systems, as well as thoughtful scaling factors to account for degree of engineering risk, overall complexity and scale, and maturity of the various contributing technologies and ecosystems. For innovative SIDRE systems, the variances in a sound estimate can rise quite dramatically. For all systems, there is also the complication of the rapid evolution of the underlying software technologies, which tends to reduce commensurability with historical comparables. The use of evidence-based proposals and independent expert review is also helpful at the source selection stage.

Estimates

An additional difficulty is the lack of a rational standard by which the cost estimates are judged. While there are well-used metrics for hardware, a uniform set of standards for software development is lacking, although there are candidate models such as SEER-SEM, True S, or COCOMO. Also, analyzing comparable probabilities of success should be a key element for awards. This analysis must avoid conflating engineering risk with programmatic risk and instead account for process plans (and earned value credit models) that acknowledge the reality of the engineering risk and indicate how it can be mitigated (as outlined above). Product-line and framework efforts provide significant challenges to development estimation, as do commercial, open-source, and vendor infrastructure and services. These outsourced products and services, loosely considered as COTS ("commercial off the shelf"), although often they may not be commercial or off the shelf, require further adaptation to the estimation models to account for the costs of assimilating the product/service, including integration, configuration, ongoing upgrade (and consequent adaptation to the subject system), licensing, sourcing risks, and other factors. For example, many conventional commercial components and services have refresh cycles, which may range from months to many years. The period of these cycles (and the extent of likely incompatibility) can often be anticipated on the basis of industry standard practices, but it nonetheless needs to be

[16] See reports from the Government Accountability Office including, GAO, 2004, *Defense Acquisitions: Stronger Management Practices Are Needed to Improve DoD's Software-Intensive Weapon Acquisitions,* GAO-04-393, Washington, DC: U.S. Printing Office.

understood, along with the prospects for "support engagements" from vendors to address urgent and critical issues should they arise.

Recommendation 2-2: The DoD should take steps to accumulate high-quality data regarding project management experience and technology choices that can be used to inform cost estimation models, particularly as they apply to innovative software development projects.

The current upgrade underway of the reporting quantities and guidance for the DI-MGMT-81739 and -81740 Software Resource Data Reports is a good example.

Contracting

There are a variety of contracting structures that are available to government program managers. Choices are generally made on the basis of goals regarding incentives for the performer. For example, the cost-plus-award-fee (CPAF) paradigm tends to front-load the incentives for performance where the product is primarily a set of artifacts that define the design, but that do not necessarily provide functional capability. It will be important to ensure that future contracts provide a balanced incentive for early development of functioning products, as well as early evaluations of performance and robustness.

An *iterative process* for software development requires somewhat of an iterative or, more precisely, an *incremental contract* with the customer, very much following the concept of a spiral model of software engineering.[17] For a company to respond to a request for proposals (RFP) with some accuracy, it generally must have experience with multiple similar projects on the basis of which it can estimate with confidence the resources and risks associated with building and testing a particular system. Some companies frequently offer a fixed-price bid as well, perhaps for as many as half of their projects, although the preferred contract is not a fixed price but rather an agreement on the general estimated figure for the cost and delivery schedule in chunks, with more specificity for the critical initial deliveries, some agreed upon process for continued negotiation around time and schedule for changes, and pricing of later parts of the system as more of it is built and delivered.

A Scenario for SIDRE Incremental Development

One possible way to combine improvement in the precision of estimation with mitigation of early-stage engineering risk (architecture, scope, hazard analysis) would be for a software customer to start a project with an initial scoping and prototyping engagement, lasting a few weeks, depending on the size or complexity of the system. This can serve to determine the scope of requirements, assess architecture alternatives, identify constituent ecosystems, and address other potential sources of up-front engineering risk. This affords both the customer and the bidder opportunity to develop more precise (but still crude) estimates of the cost and time potentially required for the project.

This scoping and prototyping phase can be used to identify what are the essential features of the system that the customer must have, what are the lower-priority features, and what are the features or functions that must be built first for the work to proceed. The company (and the customer) can then generate estimates for this first phase of the development work. This can be viewed as developing an immature design, but through a mature design process that will eventually lead to a well-validated mature design. As this initial phase of the work nears its completion milestone, systems engineers, architects, and requirements engineers can then work on a more detailed plan for the next milestone, including more specific plans for value measurement that would be used to enhance a baseline overall earned value model. The company, if it has experience with similar projects, can use historical data to adjust its estimates and add buffer time in the schedule, which will also add costs for manpower.

[17] Barry Boehm, 1986, "A Spiral Model of Software Development and Enhancement," *Communication of the ACM* 11(4):14-24.

In short, the scenario is for the customer to pay for an upfront scoping and prototyping exercise, agree to a general budget and timeframe, and then pay in increments as the work proceeds and changes. It is critical for the development team to be working extremely closely with the customer such as through having weekly or biweekly project updates and sharing information regarding architecture, features, and quality attributes of the evolving system in frequent increments. This also affords opportunity for risk mitigation regarding validation for critical requirements, enabling operational acceptance and providing evaluators an opportunity to mitigate the engineering risks they face regarding various kinds of evaluation criteria. Regarding budget, it enables the customer to achieve a budget target with essential features of the system completed and to establish options regarding additional features or quality improvements earlier in the lifecycle, thus facilitating negotiations regarding lower-priority features or bug-fixing time later on in the project.

The committee notes that real-world project experience has shown time and again that it is the early phases that make or break a project. It is therefore of paramount importance to have a strong start-up team for the early planning and architecture activities. If these early phases are done right with good teams, projects can more often be completed successfully with (stable) nominal teams of capable developers evolving the applications into the final product. If the early planning and architecture phases are not performed adequately, however, then programmatic risks escalate dramatically—even tremendous expertise may not succeed in overcoming the consequences of early bad decisions.

The committee also notes that for the largest and most complex systems, and also for many of the more innovative systems, the DoD has a strong and direct interest in architecture definition in the early project phases. DoD interests in architecture bear on longer-range issues such as interoperability, flexibility, and shifts in quality attributes as infrastructure and associated ecosystems evolve. This implies that the DoD must have capability to assess architectural decisions at the early stages as part of the overall process. There is a challenge in finding the right balance—on the one hand, contractors must fully "buy in" to architecture designs with respect to owning responsibility for outcomes, but, on the other hand, the DoD and contractors must be able to collaborate in refactoring or adapting architectures when required.

Finding 2-5: Architectural expertise is becoming dramatically more important for the DoD, its advisors, and its contractors. There will be significant and immediate benefits from advances in the state of technical support for architecture.

Recommendation 2-3: Update procurement, contracting, and governance methods to include an early and explicit architecture phase that reduces the predominant uncertainties in software-intensive systems.

Technical support for architecture includes architecture development, modeling, simulation, evaluation of quality attributes (such as performance and security), evaluation of structural attributes (such as code compliance, modularity, etc.), and techniques for adaptation. This also includes capture of architectural experience to support building on experience.

Recommendation 2-4: Define architectural leadership roles for major SIDRE projects and provide program managers with channels for architectural expertise.

With respect to risk management, if a project is structured in short cycles or milestones, such as every 4 or 8 weeks,[18] then estimates and teams can be adjusted to try to make up time on the schedule. For example, if part of the system is proving to be more difficult than planned to build or to test, then

[18] Agile cycles are typically 30 days, with a deliberate commitment for schedule-driven milestones to provide the dominant constraining structure in the management of process.

it may be possible to restructure the work plan to enable switching of people from different tasks without running afoul of Brooks's Law.[19] Mitigating risk from a contractual perspective requires reducing development cycles, system testing intervals, and feedback opportunities with the customer. Although this would vary based on scale, it would typically change release cycle times from units of months to units of weeks. This is predicated on identifying the most useful observables to support effective decision making in the feedback loop implemented in project iterations. Project managers should also be identifying early on what parts of the system have high engineering risk—such as complex components that are different from systems they have built successfully in the past.

The use of evolutionary acquisition as emphasized in DoD Instruction (DoDI) 5000.2 implies the need for continuing architectural adjustments to accommodate changing priorities, independently evolving external interfaces, new releases of COTS products, and termination of support of older COTS releases. This will be discussed next.

REALIZING DOD SOFTWARE BENEFITS VIA DOD INSTRUCTION 5000.02 AND EVOLUTIONARY ACQUISITION

As discussed above, recent DoD policy in DoDI 5000.02 has established the concept that "evolutionary acquisition" is the recommended way to acquire DoD systems, but the policy does not provide detail about how successful evolutionary acquisition can be achieved, particularly in the software arena, and in a way that is compatible with the concepts of incremental iterative development. The issue is that evolutionary acquisition requires "a militarily useful and supportable operational capability" (DoD Instruction 5000.2, p. 13, 2.c.) at each iteration, whereas incremental iterative development does not (and should not) require operational capability at every iteration. This is because the iterations in incremental iterative development may be focused on discharging particular engineering risks rather than on manifesting operational capability. Further, DoD projects currently preparing to apply evolutionary acquisition find that much of the available acquisition infrastructure (contract forms, exhibits, and data item descriptions for reviews and audits, work breakdown structures, requirements, design, test, milestone pass/fail criteria, progress payments, award fees, etc.) is still oriented around a model of single-step development to prespecified full-system requirements, with portions pre-allocated to software.

The usual result is a hardware-driven functional-hierarchy system architecture that is incompatible with preferred layered, service-oriented software architectures, and accompanying hardware-oriented work breakdown structures that encourage software suboptimization[20] and translate into management structures that hinder rapid software adaptation to change.[21] Further, projects are often unaware that there are several forms of evolutionary acquisition and choose a form that is poorly matched to their project situation. Some initial work has been done to determine the various forms of evolutionary acquisition and to provide top-level criteria for choosing among them, as shown in Box 2.2.

This top-level guidance is a good first step, but it needs considerably more detailed guidance and associated methods and tools to ensure its successful application on DoD projects.[22] What is most sorely needed at this point is an elaboration of the necessary guidance to ensure early software participation in

[19] Brooks's Law states that adding people to troubled software projects only puts them further behind schedule. See Fred Brooks, 1975, *The Mythical Man Month: Essays on Software Engineering*, Reading: Addison-Wesley.

[20] See the current revision of MIL-STD-881.

[21] Barry Boehm, A. Winsor Brown, Victor Basili, and Richard Turner, "Spiral Acquisition of Software-Intensive System of Systems," *CrossTalk*, May 2004: 4-9.

[22] Specific practices for incremental iterative development are discussed in several studies, including DSB, 2009, *Report of the Defense Science Board Task Force on Department of Defense Policies and Procedures for the Acquisition of Information Technology*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology and Logistics. Available online at http://www.acq.osd.mil/dsb/reports/ADA498375.pdf. Last accessed August 20, 2010. See also NRC, 2010, *Achieving Effective Acquisition of Information Technology in the Department of Defense*, National Academies Press, Washington, DC. Available at at http://www.nap.edu/catalog.php?record_id=12823. Last accessed August 20, 2010. The practices are also elaborated in Congressional language in the National Defense Authorization Act 2010, Section 804. National Defense Authorization Act (NDAA) of Fiscal Year 2010,

systems engineering and criteria for evaluating whether adequate evidence of software feasibility has been produced at major DoD acquisition milestones. A particular need is for guidance on stabilizing the current increment of evolutionary development while concurrently evolving the software and system architecture and plans to enable stabilized development of the next increment.[23]

Recommendation 2-5: Develop the technical and management infrastructure necessary to simultaneously support stabilized, high-assurance development of the current evolutionary increment while concurrently evolving the plans and specifications for stabilized development of the next high-assurance increment.

INTRINSIC DOD SOFTWARE EXPERTISE—BEING A SMART CUSTOMER

The Current State of DoD Software Expertise

It is widely acknowledged, including within the DoD, that the department does not have sufficient organic personnel with the software expertise to meet its needs for today's more software-intensive programs.[24] Although the DoD develops some software internally, the committe's focus here is on access to expertise that is needed for the DoD to be effective as a savvy and outstanding customer for software. This includes the expertise to effectively purchase the larger and less precedented systems as well as the precedented systems for which sensitivity to issues such as the choice of ecosystem is key. The necessary expertise includes understanding of process, architecture, requirements, and assurance. It also includes understanding of the trajectories and adoption trends for both the major commercial ecosystems and any involved DoD-intrinsic software ecosystems. The DoD faces challenges in attracting and retaining software and systems engineering personnel and also in keeping up to date the skills of the personnel they do have.[25] Commercial industry also faces challenges because demand for software expertise is high and the competition for top project managers and top architects can be particularly fierce because these two skills are both critical to success and their ranks are few.

Challenges Particular to the DoD

The defense environment poses further challenges, notably the difficulty in competing with industry to hire the most capable software architects and other experts. This is not simply a matter of salaries. For instance, it is noted by the committee that many software engineers and architects become frustrated and discouraged working within the constraints of the DoD acquisition process and with the tendency toward calcification of their "hands-on" skills that made them valuable to the DoD acquisition process in the first place.[26] Especially in recent years, the DoD has not shown the desire or ability to develop

Pub. L. no. 111-84, 111 Congress, (2009). Available online at http://www.wifcon.com/dodauth10/dod10_804.htm Last accessed August 20, 2010.

[23] See related discussion in Chapter 4.

[24] "The quantity and quality of software engineering expertise is insufficient to meet the demands of government and the defense industry." Excerpted from presentation by Kristin Baldwin, 2008, "DoD Software Engineering and System Assurance," January 15, 2008, p. 4. Available online at http://www.acq.osd.mil/se/briefs/2008-01-15-SSA-Boeing-Interchange.pdf. Last accessed August 18, 2010.

[25] Matthew Weigelt, 2009, "Officials Wants Their Own Software Engineering Experts, But They Don't Want to Disregard Industry's Experts," *Federal Computer Week.* Available online at http://fcw.com/Articles/2009/07/09/DOD-IT-systems-engineers-outsourcing.aspx. Last accessed August 20, 2010.

[26] The committee did note that federally funded research and development centers (FFRDCs) and labs can provide the opportunity to technical staff to take breaks from direct support and move to programs under acquisition. These breaks enable staff to pursue research and re-connect with their "hands-on" skills that made them valuable to the DoD acquisition process in the first place. It keeps their skills current and allows them to cycle back to another acquisition activity with fresh thoughts and approaches to developing DoD capabilities.

BOX 2.2
Software Risks

The phrase "software risks" often appears in discussions regarding software development projects and software-intensive systems engineering projects. It suggests danger and something that should be prevented or avoided by project managers. But in fact there are different kinds of risks, and not all of them involve danger. Indeed, some have an appropriate and sometimes valuable place in any innovative engineering process and its management. Most importantly, by acknowledging and managing the various categories of risks early in the process, particularly engineering risks but also system risks, overall risks related to both the engineering process and the product it develops are reduced. Differences in software risks characterize the difference between the development of precedented (routine) capabilities and unprecedented (innovative) system capabilities. Differences in different kinds of software risks characterize the difference between "critical systems" and other systems.

Risk, generally speaking, is a product of the probability of occurrence of a consequence with the degree of severity or cost of the consequence. Risk can be reduced by reducing the probability or by lessening the extent or severity of consequence or both. There are different types of risk; for software categorization is in terms of programmatic, system, and engineering risk. (Box 2.1 describes each form of risk in detail.) There are often tradeoffs and interactions among these risks.

An example is response time. To illustrate the differences and interactions, consider an example relating to a decision regarding response time of a system—for example, how frequently the tracks of enemy and friendly units are updated on a display. A longer response time may enable designers to employ precedented infrastructure and other architectural elements, yielding a more predictable engineering process. That is, from the perspective of the planning phases, a mostly linear plan to engineer the product is more likely to yield a successful outcome. In other words, programmatic risks (or project risks) are low.

The long response time may, however, create operational difficulties due to insufficient timeliness. This is a kind of system risk—the possibility of a system failing to accomplish its mission. That is, while there may be low risk in producing a system with a long response time, it may be less likely to be operationally valuable. More generally, system risks can pertain to a wide range of hazards and suitability factors in operations, such as performance, security, usability, valid functionality, and integration and interoperation. System risks can also include "long-tail" risks—events with high consequence and low (perceived) likelihood. In this latter case it can be difficult to assess how much effort should be applied to mitigate the risk.

Suppose, on the basis of up-front user studies, it is decided to require a guarantee of a specific short response time. This would certainly reduce the system risk related to suitability of the response time. But the short response time may preclude use of the commodity infrastructure and, in the absence of validated alternatives, create uncertainties in the engineering phases of the project regarding architectural choices. The resulting uncertainties and consequences created within the engineering process are engineering risks.

Which is the correct architectural choice to make? If the answer is not known until the system is put into an operational environment for test and evaluation, then the uncertainty persists for a longer period, more engineering investment is made prior to resolution of the uncertainty, and more rework is required should the choice need to be revised. Additionally, when one possibility is eliminated, uncertainty may remain regarding the choice among the remaining candidate options, and further effort may be required to resolve this choice. This adds to engineering risk, and it may add to project risk as well if there is insufficient allowance in cost and schedule for rework in the project budget. In many cases, the costs of unwinding previous bad decisions become prohibitive, and as a consequence the mismatched architecture (or other aspects of the system design) becomes a legacy infliction that is constantly worked around, adding to downstream costs and risks.

Evaluation of architectural alternatives through full development and operational tests is rarely re-

quired, however. Techniques such as architectural modeling and simulation, for example, would enable the architectural alternatives to be evaluated earlier in the process and at lower cost, lowering engineering risk. (See Chapter 5 for a discussion of the associated research challenges.)

*The probabilistic models for risk assessment have limitation*s. A software manager may find it tempting, when considering the mathematical characterization of risk as the product of consequence and probability, to develop mathematical models for probabilities. This is sometime useful but also can be dangerous, since probabilistic models often fail us in software. For example, a security vulnerability could be perceived as high consequence, but very low likelihood, and so may be left unaddressed. But once the vulnerability becomes known to adversaries (e.g., as a zero-day vulnerability), then the probability can rise dramatically, and with it the extent of risk. Unfortunately, probabilistic models fail also because of aspects other than security. The possibility of intermittent problems such as deadlocks, for example, can change quite dramatically with changes in processor, communication, or storage infrastructure. Additionally, traditional models of redundancy as a means to reduce risk are most effective when event probabilities are not coupled. But this proves to be a dangerous expectation in the engineering of software.[1]

On the other hand, in systems engineering efforts where software is embedded as part of a cyber-physical system, there are abundant probabilistic models for faults in attached physical components, and these models may have dependencies on other probabilistic models relating to aspects of the operating environment. In these cases, the familiar engineering mathematics for reliability must be employed, and the results of these analyses will inform the design of software to support tolerance or containment of errors resulting from faults in the attached components.

Credit for engineering risk reduction. The Apollo moon missions of the 1960s had systems risk (hazard) related to delivering and returning astronauts safely. This risk could be mitigated through various safety mechanisms. In general, there may be little correlation between system risks and the other kinds of risks, especially when the systems risk derives primarily from the context of operational use—system risks may be much more dependent on characteristics of the operating environment than on precedent regarding engineering decisions. But in the case of the Apollo missions, there was also considerable engineering risk, particularly early in the process when basic decisions were being made and experimentation and prototyping was being done to achieve early validation (i.e., prior to operational use) of the decisions made.

The experience of the prior Mercury and Gemini missions created precedent for many design considerations and so served to discharge certain engineering risks. In addition to relying on hard-won experience with prior systems, the principal approaches to mitigation of engineering risks involve incremental development, prototyping, and modeling and simulation. These methods reduce the cost of consequence through early feedback and response afforded.

For innovative projects, efforts to resolve engineering risks can be a significant component of overall project progress, and therefore in an earned value measurement regime there need to be ways to "give credit" for identification and discharge of critical engineering risks. This can be a challenge: How, for example, can the value to Apollo of the experience of Mercury and Gemini be weighed? Or, at a much smaller scale, how can the value of the agile practice of ongoing refactoring be assessed at a time when the costs are incurred? The refactoring practice enables teams to retain ongoing control over architectural decisions and to enhance the potential for architecture-level adaptation on the basis of future needs. But the benefits associated with the refactoring costs may appear only in later cycles, perhaps several months later, and until then the return on the refactoring investment may be difficult to assess despite the long-term value to the project.

---

[1] See, e.g., Susan Brilliant, John Knight, and Nancy Leveson, 1989, "The Consistent Comparison Problem in N-Version Programming," *IEEE Transactions on Software Engineering* 15(11):1481-1485.

and retain top technical experts within its own ranks, both in the military and civilian, except in very particular circumstances.[27] Additionally, as discussed below, it has historically proven challenging for those software experts who have remained within the DoD to maintain strong technical currency on an ongoing basis. Indeed, the committee believes that the extent of software expertise within the DoD is shrinking both relative to that of the commercial sector, and perhaps also in absolute terms.

The false perceptions that software and IT generally are reaching a plateau may lead to erroneous conclusions that the DoD can fully delegate such leadership into its supply chain. This is inconsistent with the reality of the rapid ongoing growth of software technology (as elaborated in Chapter 1) and the essential and growing importance of successful early architecture-focused decision making in the development of interlinking defense systems (as elaborated in Chapter 3).

An additional challenge to the DoD is that the split between technical and management roles will result in leaders who, on moving into management, face the prospect of losing technical excellence and currency over time. This means that their qualifications to lead in architectural decision making may diminish unless they can couple project management with ongoing architectural leadership and technical engagement. The DoD does not have strong technical career paths that build on and advance software expertise with the exception of the service labs. Upward career progression trends leading closer to senior management-focused roles and further away from technical involvement tend to stress general management rather than technical management experience. This is not necessarily the case in technology-intensive roles in industry. Many of the most senior leaders in the technology industry have technical backgrounds and continue to exercise technical roles and be engaged in technology strategy. Nonetheless, certain DoD software needs remain sufficiently complex and unique and are not covered by the commercial world, and therefore call for internal DoD software expertise. In the DoD, however, as software personnel take on more management responsibility, they have less opportunity and incentive to stay technically current. At the same time, there is an increasing need for an acquisition workforce that has a strong understanding of the challenges in systems engineering and software-intensive systems development. It is particularly critical to have program managers who understand modern software development and systems.

Commercial industry also continues to have a strong need for the same types of basic software expertise that the DoD needs and in many areas is competing with the DoD for the same pool of talent. Notwithstanding the economic downturn, salaries for personnel in these areas remain highly competitive in order to attract key talent. Although there have been improvements in recent years to accommodate highly paid technical experts, the DoD and other government pay scales remain generally not as competitive with commercial industry, making it more difficult for the DoD to attract and retain the expertise it needs. Additionally, the DoD could strengthen its ability to tap into the talent base in DoD-aligned research organizations and universities—for example, by sponsoring security clearances for technology leaders.

An additional challenge that DoD faces in obtaining and attracting key talent is the requirement for cleared U.S. citizens. Security considerations that often preclude the hiring of non-citizens markedly shrink the pool of available software talent. The pool of currently cleared U.S. citizens with the right skills is not sufficient to meet the demand, and this pool could be shrinking because of the reduction in support by the various agencies (principally the DoD, NASA, and the Department of Energy) of U.S. universities in areas related to software producibility. (The Networking and Information Technology Research and Development (NITRD) coordination categories are Software Design and Productivity (SDP) and High Confidence Software and Systems (HCSS); see Box 1.5.) University programs create the most highly qualified technical personnel, from the standpoint of pure technical expertise, which can complement DoD expertise in program management. It is the nature of university economics that

[27] See, for example, pp. 8-9 in DSB, November 2000, *Report of the Defense Science Board Task Force on Defense Software*, Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA385923&Location=U2&doc=GetTRDoc.pdf. Last accessed August 18, 2010

production of PhDs, in particular, closely tracks external research support for university projects. The recent reductions therefore mean not only that there is less U.S. research in software producibility, but also that the pipeline of software-savvy talent is diminishing. Because the DoD will not be able to directly hire the necessary talent in the short term to meet its growing needs, it needs to improve access to "DoD-aligned" talent through federally funded research and development centers (FFRDCs), Service labs, and university and industry research contractors. Flexibility regarding government personnel policies could allow more movement for leading technical experts in and out of government service, which could facilitate DoD maintenance of technical excellence and currency in rapidly changing fields.

For example, because of the rapid growth in the significance of architecture-related capability on the DoD side of major systems engineering projects, the committee has considered processes by which the DoD can gain access to the very best architectural talent to address cross-cutting architectural requirements and validation challenges. These processes include the assembly of architectural study groups and review panels of top experts, including experts drawn not just from the intrinsic DoD talent pool, but also from industry and research. Options may include focusing on trying to get engineers in mid-career in addition to young software engineers and improving the career environment so that, irrespective of age, a DoD software engineer can develop and maintain her skills by actually producing software. By bringing some software engineering work in-house, the DoD may be able to stimulate interest in DoD careers and opportunities.

The question then becomes, How does the DoD effectively become a savvy customer for these important IT and software-related services? Traditional methods have involved some combination of developing know-how internally and acquiring it from contractors. In each case, the necessary competence must be available to execute the programs, with particular emphasis on technology-intensive decision making. In much of this decision making, the DoD must define the "operating environment" for major software and systems development efforts performed by its contractors. This operating environment includes certain DoD-specific standards for interoperation, assurance, security, and so on. The expertise required in the DoD is not identical to the corresponding commercial software engineering expertise. For example, DoD large-scale software development is almost always undertaken at arm's length by contractors. This can complicate the implementation of practices that deviate significantly from the "requirements-first" RFP model. For innovative systems, as the committee has noted throughout the report, there must be ongoing interaction on topics related to architecture, incremental development, and preventive practices in support of assurance. Without appropriate expertise and experience, these interactions—and associated management of incentives—can be difficult to manage successfully. In addition, a growing number of areas of technology-intensive decision making where the DoD has particular interests and incentives vary from those of its contractors.

Access to Talent

Although access needs to be improved, the DoD does, however, have access to a considerable base of talent through three DoD-aligned sources: (1) DoD FFRDCs, (2) Service labs, and (3) research contractors in universities and industry. Despite the reduction in funding related to software (see Box 1.5), the DoD has nonetheless taken modest but valuable actions to cultivate talent and introduce leading young scientists to defense systems and the defense mission. A prominent example is the Computer Science Study Group (CSSG) sponsored by DARPA[28] that affords opportunities for younger researchers to engage more directly with defense technical challenges.

[28] For more information, see http://www.darpa.mil/dso/solicitations/ra07-43.html. Last accessed August 18, 2010.

Opportunities to Strengthen the DoD's Software Expertise

There are two significant building blocks for strengthening the DOD's intrinsic software expertise that leverage the DoD's particular expertise and responsibilities in two areas—operational test and evaluation (OT&E) and information assurance (IA). As noted elsewhere in this report, there are opportunities to expand the role of OT&E organizations to support preventive approaches to assurance and early validation, generally for innovative large-scale systems engineering projects, particularly regarding architecture, process, and key quality attributes—even when detailed decisions regarding specific aspects of functional capability are deferred. A successful IA regime will require similar engagement, as well as sophisticated interaction between defensive and offensive programs and activities, engagement with those who operate and defend the DoD's communications networks, and intelligence about threats and vulnerabilities.

In the cases of both OT&E and IA, leaders in practice and technology are understood to reside in the DoD (and in similar institutions in other countries). By creating a visible culture of elite technology-intensive leadership in these areas, the DoD has the potential to attract top talent, in a manner analogous to the ability of the National Security Agency (NSA) to attract top mathematicians. Although it is important to "grow the ranks" in these areas, the DoD cannot sustain leadership unless it recruits and engages top technical talent.

Summary

Because the DoD does not currently have the requisite expertise and talent it needs for effective software producibility and because the rapid pace of software development demands ongoing interaction with the field, the DoD must engage experts outside the DoD and its primes. This engagement, to be effective, should be accompanied by internal processes to apply and incorporate contributions and feedback to software projects throughout the systems engineering lifecycle. In other words, the DoD should adapt processes to facilitate input from outside experts throughout the systems engineering lifecycle for software-intensive systems, with particular emphasis on innovative/unprecedented and large-scale systems and on systems engineering efforts involving iterative processes.

It is essential to sponsor high-quality software-related research projects. Investing in cutting-edge software defense projects creates value not only in advancing innovation, but also in developing a pipeline of technical experts with experience tackling DoD software producibility issues. University research funding supports research opportunities for undergraduates, graduates students, and postdoctoral researchers. DoD engagement with the next generation of software experts at formative stages in their careers can encourage exploring a career within the DoD, thus increasing the available pool of cleared software professionals.

Also crucial is support for defense-relevant top-tier educational programs in U.S. universities to strengthen the pipeline of top technical experts. Targeted postdoctoral grants may be another avenue through which the DoD can encourage emerging software professionals to choose careers in the DoD.

Finding 2-6: The DoD has a growing need for software expertise, and it is not able to meet this need through intrinsic resources. Nor is it able to fully outsource this requirement to DoD primes. The DoD needs to be a smart software customer. This need is particularly significant for large-scale innovative software-intensive projects for which there are cross-cutting software architectural requirements and validation challenges.

The case for the DoD to have software expertise on its side of the table is compelling. Increasing complexity, scale, and interoperability in a context of rapid innovation and sophisticated incremental and iterative processes require the DoD to become a knowledgeable customer of software tools and

systems. Direct access to this necessary expertise, in light of industry's competing interest in hiring similar professionals, is limited. For these reasons, a combination of (1) outreach to FFRDCs and similar DoD focused organizations, academia, and industry and (2) internal DoD education and development of software expertise is needed to bridge the gap.

3

Assert DoD Architectural Leadership for Innovative Systems

The increasing complexity and scale of software systems demand that the Department of Defense (DoD) play an active role in developing and iterating systems architecture throughout the project lifecycle. This chapter characterizes the special role of architecture in software producibility, describes its particular challenges, and discusses how the DoD can strengthen its architectural leadership in software development when so much of its software development is conducted by contractors working at arm's length from DoD mission stakeholders.

SOFTWARE ARCHITECTURE AND ITS CRITICAL ROLE IN PRODUCIBILITY

Software architecture is conventionally defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them."[1,2] Just as in physical systems, architectural commitments comprise more than structural connections among components of a system. The commitments also encompass decisions regarding the principal domain abstractions to be represented in the software and how they will be represented and acted upon. The commitments also include expectations regarding performance, security, and other behavioral characteristics of the constituent components of a system, such that an overall architectural model can facilitate prediction of significant quality-related characteristics of a system that is consistent with the architectural model.

Architecture represents the earliest and often most important design decisions—those that are the hardest to change and the most critical to get right. Architecture makes it possible to structure requirements based on an understanding of what is actually possible from an engineering standpoint—and what is infeasible in the present state of technology. It provides a mechanism for communications among the stakeholders, including the infrastructure providers, and managers of other systems with requirements for interoperation. It is also the first design artifact that addresses the so-called non-functional attributes, such as performance, modifiability, reliability, and security that in turn drive the ultimate quality and capability of the system. Architecture is an important enabler of reuse and the key to

[1] Len Bass, Paul Clements, and Rick Kazman, 2003, *Software Architecture in Practice*, 2nd Ed, Boston: Addison-Wesley.

[2] There are other definitions (see http://www.sei.cmu.edu/architecture/published_definitions.html#Modern) but the principles are consistent among them.

system evolution, enabling management of future uncertainty. In this regard, architecture is the primary determiner of modularity and thus the nature and degree to which multiple design decisions can be decoupled from each other. Thus, when there are areas of likely or potential change, whether it be in system functionality, performance, infrastructure, or other areas, architecture decisions can be made to encapsulate them and so increase the extent to which the overall engineering activity is insulated from the uncertainties associated with these localized changes.[3]

Attention to the architecture is not limited to just the design and coding phases of software. Integrity of the architecture is maintained, often with supporting code analysis tools, throughout the software system lifecycle. This is done because a single software change at any stage, including maintenance in the latter stages of a system lifecycle, can violate the key architectural decision parameters essential for acceptable system behavior, for future evolution and enhancement, and for assurance.[4] During construction of a system, the architectural perspective is essential to assessing progress and risks, and the ability to make decisions and tradeoffs among various alternatives.

As systems scale up, the extent of effort that must be devoted to architecture also scales up, and slightly more steeply so that a greater percentage of overall effort is devoted to architectural considerations.[5] These include design, tradeoff analysis with respect to quality attributes from requirements, identification and analysis of precedent and related ecosystems, etc. It is noted by Boehm and Turner[6] that risk and precedent drive the balance between practices appropriate for precedented systems (i.e., "plan-driven methods") and practices appropriate for innovative systems (i.e., "agile methods"). As noted in Chapter 1, larger-scale systems most often must include both kinds of practices, especially when architectural design successfully localizes or encapsulates innovative elements and maximizes use of precedented ecosystems and infrastructures. (For further discussion of architecture, see Box 3.1.)

As also noted in Chapter 1, precedented systems are those systems whose capabilities and attributes are highly similar to those that have been produced before and therefore do not require significant software innovation. In these cases, from the standpoint of engineering risks, the most critical precedents are not of requirements, but of architecture—whenever possible, the software architecture should be well understood and derived from an analysis of previous instances of the architecture. The analysis should strongly influence the development of the software architecture for the new system, as should an understanding of the likely evolution of the involved ecosystems—and incremental evolution is characteristic of successful ecosystems. Major weapon and command-and-control systems may typically

---

[3] The nature of modularity and its value to business outcomes are explored in Alan MacCormack, John Rusnak and Carliss Baldwin, 2007, "The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry," *Harvard Business School Technology & Operations Mgt. Unit,* Research Paper No. 08-038. Available at SSRN http://ssrn.com/abstract=1071720. Last accessed August 20, 2010; and Carliss Baldwin and Kim B. Clark, 2000*, Design Rules, Volume 1, The Power of Modularity*, Cambridge, MA: MIT Press.

[4] One of the first studies of the consistency of modeled architectural intent and as-built reality in a very-large-scale code base was undertaken by Gail C. Murphy, 1996, "Architecture for Evolution," in Alexander L. Wolf, Anthony Finkelstein, George Spanoudakis, and Laura Vidal, eds., *Proceedings of 2nd International Software Architecture Workshop (ISAW'96)*, San Francisco: ACM, pp. 83-86. Follow-up work is reported in Martin P. Robillard and Gail C. Murphy, 2003, "FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code," Demonstration Session, *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. Portland, OR, May 2003, pp. 822-823.

[5] See Barry Boehm, Ricardo Valerdi, Eric Honour, 2008, "The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems," *Systems Engineering* (11)3:221-234; Mark W. Maier and Eberhardt Rechtin, 2000, *The Art of Systems Architecting,* 2nd Ed., Boca Raton: CRC Press; Manuel E. Sosa, Steven D. Eppinger, Craig M. Rowles, 2004, "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development," *Management Science* 50(12):1674-1689; Alan MacCormack, John Rusnak, and Carliss Y. Baldwin, 2006, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science* 52(7):1015-1030; and Manuel E. Sosa, Jürgen Mihm, and Tyson Browning, 2009, "Can We Predict the Generation of Bugs? Software Architecture and Quality in Open-Source Development," INSEAD Working Paper 2009/45/TOM.

[6] Barry Boehm and Richard Turner, 2003, "Using Risk to Balance Agile and Plan-Driven Methods," *Computer* 36(6):57-66. Available online at http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/Agile/r6057.pdf. Last accessed August 20, 2010.

BOX 3.1
Software Architecture

Software system architecture is conventionally defined as the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them. In essence, architecture is the set of organizing principles, both structural and semantic, that constrain the ways by which software elements of a system interact. A well-conceived software architecture can facilitate separation among work assignments allocated to software developers because it defines key elements of the "contracts" that regulate the intended connections among software elements.

Architectural commitments for a software system have implications throughout the lifecycle of the system. Architectural commitments can predict critical quality attributes related to performance, security, reliability, and other "non-functional" properties. As a consequence, architectural decisions are among the most important first decisions program managers and stakeholders make yet the most difficult to change or correct.

The committee's definition of software architectures includes more than just structural considerations relating to which components may interact with which other components through what kinds of "connectors."[1] Semantic architectural commitments may relate to protocols, data representation invariants, exceptional flow of control, timing properties and deadlines, concurrency and threading, pattern compliance, and other attributes. This combination of structural and semantic commitments is of benefit when

[1] David Garlan, 2003, "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events" in *Formal Methods for Software Architectures,* Pp. 1-24 in *Formal Methods for Software Architectures,* Marco Bernardo and Paola Inverardi, eds., Berlin: Springer Publishers.

contain many innovative elements, as do some DoD business systems. Nevertheless, there are very often large spans of functionality that are precedented. This means the overall system architecture is likely to include a mix of precedented and innovative structures.

In modern systems, the concept of "architecture" has broadened to include not only structural commitments (as noted in the definition at the start of this chapter), but also other design commitments that constrain and guide subsidiary design decisions, particularly decisions regarding how components of the system are meant to interact with other components. Architecture commitments open or close opportunities for future evolution and enhancement. In other words, it is risky if not impossible to evaluate different alternatives or different vendors without an understanding of the architectures implicit in what they propose.

In particular, modern software ecosystems have a set of architectural commitments at their core. Web services, for example, are structured in conventional ways with well-defined software interfaces among the components and, additionally, a number of "rules of the road" regarding what are appropriate interactions across those interfaces (examples shown in Chapter 1). Another example is modern data-intensive computing, such as done extensively at companies such as Google and Yahoo!. Much of this data-intensive computing activity shares a single relatively simple system architecture concept, called MapReduce, which is designed to address the challenges of distributed computing with enormous amounts of data. The MapReduce concept has turned out to be applicable to a very wide range of problems, with the result that there is now a growing community of users for a "big data" ecosystem focused around a set of open-source tools.[7] Despite the technical simplicity of the MapReduce meta-

[7] Hadoop, Zookeeper, HDFS and MapReduce at Apache are explained further online at http://hadoop.apache.org/.

it is transparent—that is, susceptible to modeling, analysis, and management of consistency with code. This enables software developers to more readily predict behaviors and interactions among existing and proposed system elements.

Software architecture also has implications for how users interact with software systems, in the sense that architectural commitments can regulate the responsiveness of a system to user redirection due to shifts in operational needs.

In some cases, a systems concept may be amenable to well-established architectural concepts and design ideas. The resulting system implementations are considered precedented systems because they rely on best-practice architectural designs or styles.[2] Often these designs are manifest in established software ecosystems that have emerged around similar system concepts, as in the case of web services implementations. In other cases, the proposed functionality and/or its associated quality attributes are sufficiently novel and complex that already established architectural concepts may not fully suffice. In these cases, new system architectures must be developed and validated in order to implement the proposed system concepts. This is very frequently the case in the DoD due to its unique mission and the reality of continued aggressive growth in functional and quality requirements. This report focuses primarily on the conception, design, and development of these innovative or "unprecedented" systems.

Although well-matched architecture is not a guarantee of success in system development projects, many project failures have been associated with inappropriate, late-breaking, or poorly articulated software architectures. Applying architectural design best practice can help decrease failure risks for both development projects and also (because of influence on quality attributes) for system operations.

[2] Mary Shaw and David Garlan, 1996, *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice Hall.

phor, its ecosystems include highly complex infrastructure to support reliability and robustness in the face of intermittent failures of individual components in large-scale data centers, such as commodity processors and disk drives. This enables data centers to be established with larger numbers of cheap commodity components rather than the more expensive options of highly reliable components, or of overall task-level checkpointing, etc.[8] This ecosystem shares architectural characteristics with Google's internal MapReduce ecosystem.[9]

A consequence is that architectural decision making for any particular software development project is profoundly influenced by knowledge of related ecosystems, of systems and hardware infrastructure, of available frameworks and libraries, and of previous experience with similar systems and projects. Small changes to architectural requirements can open or close opportunities to exploit rich existing ecosystems, greatly reducing both cost and risk.

[8] Such was the experience of the Aegis High Performance Distributed Computing Program HiPer-D project to replace expensive "mission-critical" single processors with a network of multiple affordable processors that can be made more survivable using techniques of distributed computing. For more information, see, for example, L. R. Welch, Binoy Ravindran, Robert D. Harrison, Leslie Madden, Michael W. Masters, and Wayne Mills, 1996, "Challenges in Engineering Distributed Shipboard Control Systems," in *Proceedings of Work-in-Progress Session of the IEEE Real-Time Systems Symposium*, December 1996, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.4454&rep=rep1&type=pdf. Last accessed August 20, 2010. See also Karen F. O'Donoghue and David T. Marlow, 1998, "The Application of NTP to Navy Platforms," *29th Annual Precise Time and Time Interval (PTTI) Meeting*, Long Beach, CA, December 1997.

[9] Jeffrey Dean and Sanjay Ghemawat, 2004, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004. Available online at http://labs.google.com/papers/mapreduce.html. Last accessed August 20, 2010.

Consider, for example, choices made by developers of a software system that involves complex interactive graphical user interaction (GUI) to visualize results stored in a large database, such as results of modeling and simulation. The architecture of this software system will be influenced by available choices for the database and the GUI frameworks. The architecture of the overall application will be guided by several factors: (1) framework design constraints implicit in the GUI frameworks available, (2) database interfaces offered by potential vendors, and (3) prior experience in developing interactive visualization systems for modeling and simulation. The architectural decisions are thus informed by experience, by knowledge of the principal ecosystems and conventional structures used for GUI frameworks and databases, by particular refinements of those conventional structures associated with candidate software components, and by infrastructure commitments.

Architecture can often be reused across similar products, encouraging development of supporting tools to facilitate its reuse. In a product-line approach, a family of related systems shares architectural elements and components. This kind of approach offers not only significant cost advantages, but also promotes faster time to market, ability to reuse infrastructure, and, perhaps most importantly, significantly reduced engineering risk (i.e., uncertainties and consequences related to the process of making, acting on, and validating design commitments during the engineering process). Indeed, sound architectural decisions, often even more than requirements decisions, are a principal enabler of product lines, of flexibility in evolution and maintenance, of ability to exploit the rapid growth in hardware infrastructure (processors, storage, communications), and of ability to build effectively on the increasingly powerful base of software infrastructure (operating systems, databases, GUI frameworks, etc).

SOFTWARE ARCHITECTURE IN INDUSTRY

Industry leaders in the development of software pay considerable attention to the software architecture as an essential, not an accidental, set of design decisions. Throughout a system's lifecycle, a managed software architecture provides critical evidence that a software-reliant system is capable of meeting its business and mission goals. Respected companies such as Microsoft, Intuit, IBM, J.P. Morgan, Bosch, and Siemens engage in rigorous training and certification processes for their architects.

In the case of Microsoft, software architects are drawn from the most senior and accomplished technical people in the company and have the responsibility for investigating architectural alternatives, designing a system's architecture, and ensuring that the resulting software product adheres to the desired architecture. At one time, the Windows team had an architecture review board composed of senior technical contributors and tasked with reviewing all groups' potential changes and extensions to Windows.

IBM has evolved corporate standards for technical roles and technical career paths for software professionals including architects. These senior positions are achieved through years of apprenticeship, a track record of accomplishment, and approval by a certification board composed of technical peers. Titles such as senior IT architect, distinguished engineer, and IBM fellow reflect highly influential roles with "executive" standing within IBM.

In addition to staffing and standards, architecture, or rather architectural reuse, plays an important role in the development of many successful commercial ecosystems and product lines. To take just one example of a product line, Apple's recently introduced iPad tablet reuses in large measure the software architecture from the iPhone. The iPhone, in turn, reuses many key architectural and infrastructural elements from systems as early as the NeXT computer, which was introduced more than two decades ago.

Many product-line commonalities exist across time, in the sense of architecture designed to anticipate evolution of capability, institutional growth in capacity, and other "natural" steps of growth. The generality of a suitable systems architecture is significant for the DoD because many of its systems persist for several decades.

But there is a danger in misconstructing the concept of "generality" in architecture. It is well estab-

lished in software practice—and a core tenant of agile techniques—that excessive structural generality at the outset of a project can create enormous inefficiencies and engineering risks, slowing down developers who must provision that generality despite the fact that it may not be needed downstream. The generality also creates added engineering risk, because it complicates the natural progression of change and evolution that is characteristic of incremental development and long-lived software systems generally. There is thus a balance that must be struck. The best "generality" may be in the form of well-crafted "seed" commitments at the outset of a project that can: (1) establish architectural direction, (2) yield the intended benefits in quality attributes and flexibility, and (3) afford engineering leaders sufficient intellectual control and flexibility to enable ongoing refactoring as required over a systems lifecycle.

Other product-line commonalities exist within subsystems or in components of larger systems—that is, "product line" is not just meaningful at the level of overall systems. As DoD systems scale up and interconnect, the architectural commonalities that comprise "product lines" could focus increasingly on designs for the common protocols that support interlinking and for the many sets of data formats and representations that are needed to permit information flow among DoD and other systems. The committee characterize these as "architectural" commonalities because they serve in the role described at the outset of this chapter—as key design commitments that support quality attributes and that constrain implementation choices.

Finding 3-1: Industry leaders attend to software architecture as a first-order decision, and many follow a product-line strategy based on commitment to the most essential common software architectural elements and ecosystem structures.

Note that this finding focuses on the most essential commitments as comprising initial architectural decisions—more is not necessarily better.

ARCHITECTURAL PROBLEMS AS A SOURCE OF SOFTWARE PROBLEMS

Whether or not it is explicitly identified and managed, every software system has an architecture, and larger systems have multiple levels of architecture definition, addressing design choices regarding subsystems and components. Although having a well-matched architecture is not a guarantee of success, software systems that are not based on well-formulated software architectures are more likely to exhibit the kind of software horror stories too often experienced in DoD acquisitions with respect to project risk.[10] At the product level, with respect to systems risk, these are the systems with communications bottlenecks, systems that hang up or crash, systems that have difficulty re-synchronizing after disconnect, systems with database access that is sluggish or unpredictable, and systems that users judge as overly complex.

These horror stories also occur in previously well-performing legacy systems after a maintenance release, often because code changes violate the architecture, which in many cases was not explicit and therefore not managed. At the process level, these are the systems that are unable to exploit established and evolving ecosystem infrastructure and improving software component capabilities, that cannot readily interoperate and federate with other systems, that defy effective quality evaluation practices, and that iterate in development without convergence (so-called "death spirals"). Perhaps most significantly, these are the systems whose engineering risks and uncertainties most often fail to resolve and

[10] See Daniel L. Dvorak, 2009, "NASA Study on Flight Software Complexity," *Technical Report, AIAA Infotech@Aerospace Conference*, April 6-9, 2009, Seattle: American Institute of Aeronautics and Astronautics, Inc.; see also J. Elm, D. Goldenson, K. Emam, K. Donatelli, and A. Neisa, NDIA SE Effectiveness Committee, 2008, *A Survey of Systems Engineering Effectiveness*, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2008-SR-034, available online at http://www.sei.cmu.edu/reports/08sr034.pdf, last accessed August 20, 2010. See also NRC, Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds. *Software for Dependable Systems: Sufficient Evidence*? Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11923. Last accessed August 10, 2010.

so are transformed into system and project risks (Chapter 2). Recent Studies by the Defense Science Board (DSB) and NASA confirm that software architectural issues are identified as a systemic cause of software problems.[11,12]

THE DOD EXPERIENCE WITH ARCHITECTURE-BASED DEVELOPMENT

The DoD experience with long-term software acquisition programs has provided strong evidence for the value of software architecture. The Air Force Command Center Processing and Display System-Replacement (CCPDS-R) program[13] demonstrated the benefits of architecture-based development and the value of supporting tools, even when developed for a single program.[14] In addition, the 1994 DSB[15] study observed that for many DoD systems, both the functionality and the non-functional attribute requirements were similar to previously acquired systems, and could be subjected to the kind of analysis that the Air Force Electronic Systems Division (ESD)[16] performed on Command Centers. In a demonstration program called PRISM (Portable Reusable Integrated Software Module), ESD analyzed the Command Centers that it had helped acquire over the preceding decade. Based on that analysis, a common architecture was developed that could be tailored to accommodate unique requirements so that ESD could support a product line for a class of command centers.[17]

ESD also recognized that such common architecture could be supported by a collection of tools, including code generation, that as a suite significantly reduced the risk, cost, and time required to acquire a new command center. In this manner, success in innovation in early designs yields explicit precedent, in the form of architectural successes, that can reduce at-the-margin costs and risks for adding new command centers and incremental new functionalities. A similar demonstration effort was supported by the Air Force Aeronautical Systems Division (ASD) for development of a prototype architecture for aircraft simulators.[18] That demonstration showed that the Air Force was able to acquire simulators for different

---

[11] Defense Science Board (DSB), 2009, *Report of the Defense Science Board Task Force Department of Defense Policies and Procedures for the Acquisition of Information Technology*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at https://www.stk.com/downloads/resources/download/risk-and-cost-reduction/2009-04-IT_Acquisition_New.pdf. Last accessed August 20, 2010.

[12] Daniel L. Dvorak, ed., March 2009, "NASA Study on Flight Software Complexity," Jet Propulsion Laboratory California Institute of Technology, Pasadena, CA. Available online at http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf. Last accessed August 20, 2010.

[13] "The Air Force's Command Center Processing and Display System Replacement (CCPDSR) program provides another reuse variation. TRW, the prime contractor, took software developed and funded under the CCPDSR contract, and updated and reworked the product using internal funds, with the intention of selling it commercially. TRW was successful and has since licensed it, under the acronym UNAS (Universal Network Architecture Services), to both Digital Equipment Corporation and Rational. … Clearly, this reuse occurred through TRW's initiative, and has been commercially successful. Other applications which may benefit from work done under CCPDSR include ATCCS and the Air Force's Systems Software and Design Center." Quoted from Unisys Corporation, March 1991, *US45 - Current FAR and Budget/Finance Requirements*, Reston, VA. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA240917&Location=U2&doc=GetTRDoc.pdf. Last accessed August 20, 2010.

[14] Walker E. Royce, 1998, *Software Project Management: A Unified Framework*, Reading: Addison Wesley.

[15] DSB, 1994, *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics.Available online at http://www.dod.gov/pubs/foi/reading_room/859.pdf. Last accessed August 20, 2010.

[16] ESD and ASD were part of the Air Force Systems Command (AFSC) until 1992 when the Air Force Materiel Command was established.

[17] Randall W. Lichota, Robert L. Vesprini, Bruce Swanson, 1997, "PRISM Product Examination Process for Component Based Development," *5th International Symposium on Assessment of Software Tools (SAST '97)*, Pittsburgh, PA, June 3-5, p. 61.

[18] William K. McQuay, 1997, "Air Force Modeling and Simulation Trends: Modeling and Simulation Makes Possible the Unaffordable." *Program Manager Magazine*, September-October 1997: 128-132. Available online at http://www.dau.mil/pubscats/PubsCats/PM/articles97/ms_usaf.pdf. Last accessed August 20, 2010. See also William K. McQuay, 1996, "Modeling and simulation trends and J-MASS technology," *Proceedings of the IEEE 1996 National Aerospace and Electronics Conference (NAECON)*,May 20-23,1996 , Dayton OH, pp. 579-584. Available online at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=517707&isnumber=11050. Last accessed August 20, 2010.

aircraft using the same base architecture at significantly lower cost and on an accelerated schedule, even in parallel with the aircraft development. Similarly, tools were developed to support the effort.

These demonstration projects were originally conducted in the late 1980s and promised to put the DoD in a leadership position with respect to software architecture-driven acquisition. Based on these promising demonstration efforts and similar experience among industry leaders, the DSB recommended that the DoD institute an architecture-driven acquisition strategy for software-intensive systems.[19] The DSB also recommended that other application areas be similarly analyzed and common software architectures developed to establish the basis of a product line.

Unfortunately, these demonstration programs were not continued. This was largely for the reason that the DSB has cited as a primary inhibitor—namely that each program is autonomous and believes that its requirements are somehow unique, and thus conformance with precedent could be perceived as excessive compromise. The reality is that such conformance is a principal pattern of successful commercial innovation, even for the most aggressive projects. The study recommended that the Program Executive Offices (PEOs) be given funding and staffing to perform such analyses and develop or acquire appropriate software architectures. That advice has been repeated by the DSB in other studies. The benefits of such approaches are proven, significant, and well documented by industry case studies across multiple domains.[20]

One of the challenges to success is the capability of the PEO organization to make the technical case for a set of architectural decisions that constrain the decision space of program offices and primes. These constraints provide broad advantage across a family of systems and for particular system development efforts, with respect to precedent and risk. Chapter 2 considers the nature of engineering risk and the considerable benefits of reducing it by following precedented architectural pathways. Unfortunately, those benefits (or architectural reuse) may not be easily measured at the outset, while at the same time designers and developers may have concrete complaints over the architectural constraints imposed (also at the outset) so that those downstream benefits can be realized. Additionally, contractors (and government program managers) may not always be offered appropriate incentives. The natural inclination is to develop new architectures and infrastructural elements rather than compromise some autonomy, accept modest near-term engineering risk (in exchange for mitigation of major long-term engineering risk), and implement a bias toward adopting existing infrastructure, product-line architectures, and other ecosystem models.[21]

Another trend within the DoD is to find ways to accelerate fielding of new capabilities. One idea beginning to get traction is to focus on getting 80 percent of the requirement fulfilled in significantly less time and cost.[22] One enabler of rapid deployment with reduced risk is to base the development on known software architectures that are appropriate to the application or to develop architectures such that they support incremental releases while still providing persistent quality behavior. Although there are intrinsic incentives in the commercial product space, as noted above, those incentives may not always be present in system acquisition efforts, unless there is appropriate planning at the outset.

---

[19] DSB, 1994, *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, Washington, DC. Available online at http://www.dod.gov/pubs/foi/reading_room/859.pdf. Last accessed August 20, 2010.

[20] "Catalog of Software Product Lines" Sofware Engineering Institute.edu http://www.sei.cmu.edu/productlines/plp_catalog.html. Last accessed August 20, 2010

[21] Several of the ideas mentioned here and in Chapter 2 are supported in Section 804 of the 2010 National Defense Authorization Act. National Defense Authorization Act (NDAA) of Fiscal Year 2010, Pub. L. no. 111-84, 111 Congress, (2009). Available online at http://www.wifcon.com/dodauth10/dod10_804.htm. Last accessed August 10, 2010.

[22] DSB found that "Program management does not encourage 80% solution for 20% cost." On p. 23 in DSB, June 1994, *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*, Washington, DC: Office of the Under Secretary of Defense for Acquisition and Technology. Available online at http://www.dod.gov/pubs/foi/reading_room/859.pdf. Last accessed August 10, 2010.

Finding 3-2: The technology for definition and management of software architecture is sufficiently mature, with widespread adoption in industry. These approaches are ready for adoption by the DoD, assuming that a framework of incentives can be created in acquisition and development efforts.

Finding 3-3: The DoD would benefit from explicit attention to software architecture and industry best practice, including (1) formalizing career paths and role descriptions for software architects, (2) identifying ways that DoD-aligned software architects can provide objective advice (see Chapter 2), and (3) enhancing organizational structures to support effective architectural leadership.

This finding applies to both precedented and innovative ecosystems. Architecture is increasing in importance due to the broadening role of software in systems, the increasing interlinking of systems, and the growing role of ecosystems and the consequent growth in access to common software infrastructure.

Despite the organizational presence of PEOs, the committee has unfortunately not seen evidence that the DoD has moved toward an overall acquisition strategy for innovative software-intensive systems in which software architecture has a principal role. As noted above, such approaches can make sense for several possible reasons: First, where software requirements for multiple systems are similar, software architectural commitments enable product-line strategies, with the benefits not only of reuse of common infrastructure, but also of reduced engineering risk because the reuse is planned. A bias toward commonality across similar systems is the means by which new software ecosystems are created (Chapter 1). The resulting benefits in cost can be very significant—sometimes an order-of-magnitude reduction. A second rationale for early architectural commitment is planning for interoperation. Indeed, many of the identified post-deployment difficulties with interoperation and platform evolution are symptoms of insufficient planning with respect to requirements and architecture. A third benefit, very important for management of costs and engineering risk in long-lived defense systems, is planning for flexibility—architecture commitments effectively define and encapsulate areas where change is anticipated, or not (more on this aspect below).

Admittedly, at the time of the initial 1994 DSB recommendation,[23] software architecture was not as well understood, and supporting practices and technology not as well developed, as they are today. In the ensuing 10 years, a significant body of work has been amassed that validates these recommendations and codifies best practices. Books have been written about software architecture, software architecture training is available, universities offer courses, software architecture assessment methodologies are available, and code validation tools are available to verify consistency with the architecture.

Finding 3-4: Several DoD programs are using software architecture-driven acquisition with successful results.

There are programs that followed an acquisition strategy driven by early commitments regarding software architecture and that illustrate the benefits that would be obtained from a pervasive commitment to an architecture-driven approach. For example, the Army Integrated Battle Command System (IBCS), the Air Force Joint Mission Planning System (JMPS) and the Navy Common Link Integration Processing (CLIP) Program had architecture-driven approaches written into the request for proposals (RFP) and contract language. Software architecture played a major role in the RFP and source-selection activities in the Navy DDG-1000 Program. The Army Warfighter Information Network-Tactical (WIN-T) has applied an architecture-centric approach in two different acquisition phases after the contract was

[23] DSB, November 1994, *Summer Study on Information Architecture for the Battlefield*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology. and Logistics. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA286745&Location=U2&doc=GetTRDoc.pdf. Last accessed August 20, 2010.

in place and has conducted both software architecture and system architecture evaluations using the Architecture Tradeoff Analysis Method (ATAM).[24]

The Air Force GPS-OCX Program is also pursuing an architecture-driven approach. Recognizing the importance of the software architecture and providing for contractual means to ensure appropriate software architecture practices and artifacts have provided early, objective evidence about software structure and system behavior. Moreover the Air Force has established baselines against which to manage software development over the lifecycle. Such efforts, when supported with appropriate analysis tools, identify risks early in the lifecycle and, when such risks are mitigated, significantly reduce overall program risk.

The Army's Strategic Software Improvement Program (ASSIP), which seeks to improve the way in which the Army acquires software-intensive systems, established an architecture initiative to increase the focus on software architecture as part of major systems acquisitions and to develop organic capability within the Army for architecture-centric practices.[25] One study of 12 major systems indicated successful use of architecture practices, in particular architecture evaluation, to increase understanding of system requirements, design decisions, and risks.[26] On average, 25 significant risks are found during a software architecture evaluation. For example, a recent use of the ATAM coupled with an Architecture Analysis and Design Language (AADL)[27] uncovered previously undiagnosed design flaws in Apache's new runtime system.[28]

Although it is difficult to place a quantitative value on early-detected technical risks, in a study performed by the National Defense Industrial Association (NDIA) in 2006-2007, which surveyed 64 programs and projects that had defense contractors, 46 percent of the projects that had higher product (software) architecture capability delivered the best project performance. In comparison, only 11 percent of the projects that had the least architecture capability delivered the best project performance.[29] This study concluded that the early phases of systems engineering, which include software architecture development, have the most impact.

When an architecture-driven approach is taken to establish a software product line, the quantitative impact is well documented. At this juncture, there are both government and defense industry examples including Army Training Support Center (ATSC); Advanced Multiplex Test System (AMTS); Army's Common Avionics Architecture System (CAAS) Product Line;[30] Textron Overwatch Intelligence Center Software Product Line;[31] the Live, Virtual, Constructive Integrating Architecture (LVCIA) product lines

[24] "Software Architecture and Tradeoff Analysis Method," available online at http://www.sei.cmu.edu/architecture/consulting/systematam/index.cfm. Last accessed February 20, 2010.

[25] Mark Kasunic, 2004, *Army Strategic Software Improvement Program (ASSIP) Survey of Army Acquisition Managers, Technical Report*, CMU/SEI-2004-TR-003, Pittsburgh, PA: Carnegie Mellon University/SEI. Available online at http://www.sei.cmu.edu/library/abstracts/reports/04tr003.cfm. Last accessed August 20, 2010.

[26] Robert Nord, John K. Bergey, Stephen Blanchette, Jr., and Mark H. Klein, April 2009, *Impact of Army Evaluations*, Pittsburgh, PA: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/library/abstracts/reports/09sr007.cfm. Last accessed August 20, 2010.

[27] "Architecture Analysis and Design Language," available online at http://www.sei.cmu.edu/dependability/tools/aadl/index.cfm. Last accessed August 20, 2010.

[28] Peter H. Feiler and Dionisio de Niz, 2008, *ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System, Engineering Practices, Special Report*, CMU/SEI-2008-SR-001, Pittsburgh, PA: Carnegie Mellon University/SEI. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA480129. Last accessed August 20, 2010.

[29] Joseph P. Elm, Dennis Goldenson, Khaled El Emam, Nichole Donitelli, Angelica Neisa, and NDIA SE Effectiveness Committee, 2008, *Survey of Systems Engineering Effectiveness— Initial Results, A, Special Report*, CMU/SEI-2008-SR-034, Pittsburgh, PA: Carnegie Mellon University/SEI. Available online at http://www.sei.cmu.edu/library/abstracts/reports/08sr034.cfm. Last accessed August 20, 2010.

[30] Paul C. Clements and John K. Bergey, 2005, *The U.S. Army's Common Avionics Architecture System (CAAS) Product Line: A Case Study, Technical Report*, CMU/SEI-2005-TR-019, Pittsburgh, PA: Carnegie Mellon University/SEI. Available online at http://www.sei.cmu.edu/library/abstracts/reports/05tr019.cfm. Last accessed August 20, 2010.

[31] Paul Jensen, 2009, "Experiences with Software Product Line Development," *CrossTalk* 22(1):11-14.

at Army PEO/STRI; and BAE's Diamond software product line.[32] As the successful case studies indicate, the government benefits through reduced engineering risk, reduced development and maintenance costs, decreased time to field, increased system agility, and improved system quality. The development of new ecosystems centered around the derived architectures also fuels the competitiveness of U.S. defense software industries.

There is the possibility that the focal point for such cross-program leverage could be through the PEO structure. But the committee perceives that the necessary authority and budget flexibility do not exist. Nonetheless, occasionally PEOs are able to accomplish such leverage.

SUPPORTING TECHNOLOGY AND RESEARCH NEEDS

A number of tools are emerging or maturing that can assist in assessing potential architectural design decisions. One of the most fundamental is the tradeoff analysis of diverse quality attributes in requirements with architectural models. An example of a systematized process for conducting this tradeoff analysis, noted above, is ATAM.[33] Another example is the Google File System (GFS) and the tradeoffs in its design among scale, cost, and reliability.[34]

In addition to this and similar process-based approaches, there are a number of significant technical enablers of architectural design. Examples of these include:

- *Systems instrumentation and profiling*. Techniques to collect data from running systems can give very significant insights into the behavior and structure of the system, and provide inputs for reconstructing the architecture "as-implemented." A principal feature of modern adaptive and self-healing architectures (sometimes called "autonomic systems") is a pervasive approach to instrumentation within a system, including at enterprise scale. The resulting instrumentation data can be used in real-time to support monitoring for security, dynamic balancing of resource usage, and reassignment of tasks in the event of local errors and failures within a large system. Additionally, the data can be used forensically to diagnose performance, reliability, and security issues. Diverse techniques can be used to analyze the data in real time, in near-real time, and forensically. These techniques range from simple rule-based pattern matching to machine-learning technologies and data-mining techniques. The multi-purpose nature of instrumentation data has the added benefit of facilitating a "return on investment (ROI) case" for inserting the instrumentation and support for storage and analysis of the resulting database.
- *Interface specification models and tools*. The specification and enforcement of protocols of interaction among heterogeneous software components include not only language-support "API" specifications, but also many additional constraints or "rules of the road" regarding protocols for interaction, preconditions on inputs, state constraints on objects, roles for threads, and so on. As we improve our ability to specify these constraints more completely, we become better able to separate the processes of developing separate components and assuring their compatibility. On the other hand, modern framework APIs are much more complex than the simpler library application programming interfaces (APIs) and protocol definitions of earlier systems. These are pervasive in web services, GUI development, AJAX rich clients, enterprise resource planning (ERP) systems, mobile frameworks, and many other areas. The advance

[32] John K. Bergey, Sholom Cohen, Patrick Donohoe, Matthew J. Fisher, Lawrence G. Jones, and Reed Little, 2009, *Software Product Lines: Report of the 2009 U.S. Army Software Product Line Workshop, Technical Report,* CMU/SEI-2009-TR-012 Pittsburgh, PA: Carnegie Mellon University/SEI. Available at http://www.sei.cmu.edu/reports/09tr012.pdf. Last accessed August 20, 2010.

[33] Rick Kazman, Mark H. Klein, and Paul C. Clements, 2000, *ATAM: Method for Architecture Evaluation, Technical Report,* CMU/SEI-2000-TR-004, Pittsburgh, PA: Carnegie Mellon University/SEI. Available at http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm. Last accessed August 20, 2010.

[34] For visualization of the GFS architecture and more discussion see Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, 2003, "The Google File System," *ACM SIGOPS Operating Systems* 37(5):29-43. Available online at http://labs.google.com/papers/gfs-sosp2003.pdf. Last accessed August 20, 2010.

of the underlying software technology both increases this complexity (as the power of frameworks increases) and also increases our ability to handle it using interface specification techniques.

- *Change impact analysis*. Artifacts such as modularity analyses and dependency matrices can be used to identify interdependencies among design decisions relating to separate components of a system.[35] When there are design decisions that are expected to change over time, architectural structures can be developed to "contain" or encapsulate those particular design decisions in an implementation such that anticipated subsequent changes influence relatively few other components. Techniques related to dependency matrices are used to assess interdependencies and, based on models of potential changes to design decisions, assess the extent of impact of potential changes. When expectations regarding change rates are known early in the process, architectures can be structured to risks associated with change and to localize the cost impact of subsequent changes. Looking down the road, a combination of change impact analysis and interfaces modeling can facilitate a more incremental approach to verification of new changes.
- *Architectural suitability measures*. In addition to considerations of Conway's Law, which observes the relationship between system structure and organizational/sourcing structure in large systems, there are also internal measures of quality for architectural designs. The most significant measures are (1) *coupling* among components, (2) *cohesion* within components, and (3) *correspondence* of changeability in requirements elements with encapsulation in architectural elements. The exact character of the measures depends highly on the specific nature of architectural models and code manifestations selected by a project. Nonetheless, these overall measures can be significant indicators of the success of an architect in achieving modularity goals.
- *Cross-cutting analyses*. Mechanisms such as architectural specification techniques, code-quality specification fragments, compile-time "pragmas," and other declarations can be used to check that dependencies disallowed in the architecture are not added to the code. These techniques can enable developers and architects to monitor the consistency of an evolving code base with architectural intent (which may also be evolving, albeit at a slower pace). There are particular families of techniques, such as aspect-oriented technologies, that, roughly speaking, facilitate better linkages and traceability between models and code. Although there are many potential perspectives on the code base for a software system, certain cross-cutting considerations, although completely precise and evident in models, cannot readily be made evident in code in a way that supports monitoring and analysis. Aspect-oriented and related technologies can provide a means to accomplish this for many kinds of models.
- *Assisted code generation*. Tools that support model-driven engineering approaches can be used to assist developers in deriving implementations from models. In some cases, source code is generated directly from high-level problem-focused specifications. In other cases, code templates and frameworks are used, which can be filled dynamically or at load time. This concept of domain-specific languages (DSLs) is directly analogous to the generation of intermediate or machine code by compilers (e.g., the Common Intermediate Language and associated infrastructure in Microsoft's .NET, or the Java Virtual Machine bytecode language in the Java ecosystem from Oracle/Sun). In particular, if generated code is edited, then there is no longer an assurance (derived from trusting a compiler) of conformance with the high-level intent. Over the long haul, techniques for code generation are an intermediate step from routinized design toward configurable automation of capabilities and conventionalized interfaces in larger systems. Indeed, there is no significant operational difference between fully automatic code generation and configurable components—both lose traceability when results are directly modified, and both pose similar challenges to verification.

[35] Carliss Baldwin and Kim Clark, 1999, *Design Rules: The Power of Modularity Volume 1*, Cambridge, MA: MIT Press. See also Carliss Baldwin and Kim Clark, 2007, "Modularity in the Design of Complex Engineering Systems," pp. 175-205 in *Complex Engineered Systems: Science Meets Technology*, Dan Braha, Ali A. Minai, and Yaneer Bar-Yam, eds. Berlin: Springer Berlin/Heidelberg.

Many of these techniques are sufficiently well established that they can be adopted by conservative and prudent program managers. Unfortunately, the up-front costs of adapting and using such techniques present an apparent budgetary risk that most program managers will not accept. In addition, in some cases these tools need to be matured and adapted for DoD systems, and that level of investment is beyond the capability of individual program managers. The DoD needs a systematic investment in identifying and maturing such tools and a means of encouraging their use.

However, the extremely rapid pace of technology development mandates constant attention to the advancement of these techniques to ensure that capabilities continue to be applicable, especially in response to the enhanced architectural needs of more complex future systems. In particular, the DoD needs to conduct the research necessary to understand the software architectural issues for DoD applications (Chapter 5). As previously observed, such research is unlikely to be supported at the program level. Also as previously mentioned, neither will it emerge from industry research efforts, which are primarily product driven.

Architecture leadership is well established in commercial industry, as noted.[36] It is also a specialty in aerospace firms. Contractual and business incentives may sometimes conflict with goals in advancing underlying technological enablers for architectural leadership, in advancing reuse, and in creating architecture-enabled mechanisms to support interlinking (interoperation across systems). In open-source projects, architectural innovation is less of a primary focus. The established successful open-source projects such as Linux, Apache, and Firefox have historically tended to build on precedented architectural concepts.

Recommendation 3-1: Initiate a targeted research program to provide software architects with better tools and techniques for DoD systems.

Chapter 5 lays out a broad range of research that the DoD must conduct to understand the software architectural issues for DoD applications. But there are significant near-term opportunities for the DoD to invest in a systematic way to improve practice. Specifically, each PEO could be appropriated a budget to support:

- Identification and analysis of existing software architectures and ecosystems for the application areas for which the PEO is responsible;
- Evaluation of the common features of those architectures leading to the definition of a product-line approach for those systems and of common architectural elements and data models across systems; and
- Development of improved architecture-based practices for future development.

[36] Industry examples can be seen in Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, 2007, "Dynamo: Amazon's Highly Available Key-value Store," *21st ACM SIGOPS Symposium on Operating Systems Principles*, W.A. Stevenson, ed., ACM, pp. 205-220. Also see an example of interface specifications—WSDL (Web Service Description Language, available online at http://www.w3.org/TR/wsdl and cutting analysis tools available at http://research.microsoft.com/apps/pubs/default.aspx?id=70226, and http://www.coverity.com/). Last accessed August 20, 2010.

STRENGTHENING DOD CAPABILITIES WITH RESPECT TO ARCHITECTURE

The committee also notes that segments of the defense industry are capable of supporting software architecture-driven acquisition and product-line strategy.[37,38,39,40] Despite a lack of financial incentives from the DoD, a number of defense industry contractors have developed a cadre of software architects, supported by training and certification processes. They state that they expect it will enable them to do a better job for programs they win and will better position them to win similar programs in the future. Also within the defense industry are corporate software product-line initiatives and successful product-line efforts that support DoD acquisitions. This implies that segments of the defense industry are well positioned to support such a move on the part of the DoD. If the DoD decides to pursue a software architecture-driven acquisition strategy followed by a product-line strategy for systems with consistent base requirements, the segments of the defense industry are equipped to respond. Further, if the DoD backs up the move to a software architecture-driven acquisition strategy with training of software architects, and supports research to develop and improve related supporting technology, the DoD could assert leadership. (See Chapter 2 for a discussion of the current state of DoD software expertise and options for increasing expertise.)

The Office of the Assistant Secretary of the Army for Acquisition, Logistics, and Technology has recently issued a mandate that each program be staffed with a software architect.[41] This is an important example of the kind of organic capability the DoD can and should develop. These architects will neither develop software architectures nor design or implement code. However, they will be trained in software architecture practices and how that relates to acquisition of software-reliant systems in the Army. The skill set they will need includes: understanding how to evaluate software architectures, having analyses available to understand which architectural decisions will be appropriate for their requirements, employing tools to ensure that code conforms to the architecture, and building on experience to manage integrity of the software architecture during system evolution. An investment by the DoD in practices and research that will support this Army initiative can help it succeed.

There are significant challenges to achieving success with an architecture-led model. These derive from the difficulty of structuring incentives and allocating/sharing risks among the key stakeholders, including the development contractor, the DoD program management organization, the ultimate operational users, and managers of related systems that might be potentially interlinked. The committee identifies here several of the challenges. One of the early issues that must be addressed involves the question of how the architecture is defined, as well as who owns the architecture. The options that have been considered[42] include the DoD separately contracting for the development and/or selection of the architecture and supporting tools through a competition, and then selecting the best bid. An issue with this approach is the sharing of risk and responsibility among the architecture developer, the prime, and the DoD regarding architectural decisions. This is a significant challenge when the DoD seeks to impose architectural constraints to enhance product-line opportunities, foster interoperation, and manage the development of ultra-scale systems where multiple prime contractors are involved.

---

[37] Lisa Brownsword, Paul C. Clements, 1996, *A Case Study in Successful Product Line Development, Technical Report, ,* CMU/SEI-96-TR-016, Pittsburgh, PA: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/library/abstracts/reports/96tr016.cfm. Last accessed August 20, 2010.

[38] David C. Sharp, 1999, "Avionics Product Line Software Architecture Flow Policies," *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Louis, MO.

[39] David Kaslow, 2000, "Architecture Based Design Applied to a Remote Sensing Satellite Planner," *INCOSE* 2000, Minneapolis, MN. Available online at http://www.incose.org/delvalley/3_4_3_abd.pdf. Last accessed August 20, 2010.

[40] Bert Schneider, Dale Anglin, Erik Baumgarten, John Dinh, and Mark Hall, 2008, "Raytheon Reference Architecture (RA): Enabling Timely & Affordable Customer Solutions," *13th ICCRTS: C2 for Complex Endeavors*.

[41] See memo from LTG N. R. Thompson dated May 26, 2009, and referenced in Bill Pollak, 2010, "Software Architects: Are You Losing Ground If You Are Not Credentialed?" *Saturn Network Blog*, February 12, 2010. Available online at http://saturnnetwork.wordpress.com/tag/architecture-tradeoff-analysis-method/. Last accessed August 20, 2010.

[42] Adapted from John K. Bergey and Wolfhart B. Goethert, 2001, *Developing a Product Line Acquisitions Strategy for a DoD Organization: A Case Study*, SEI Technical Report, Pittsburgh, PA: Carnegie Mellon University.

One idea is for contractors, or bidders, to first contribute to the architecture in the early phases of an incremental development, and then help to implement it—with contract incentives for the implementation phase tied to the actual success of the architecture. This can be challenging in a competitive environment where contractors may derive differential advantage from particular architectural commitments—and where those commitments may conflict with other goals, such as interlinking of systems.

Another challenge is to develop a cadre of architects who have suitable engineering experience—and have ongoing engineering engagement, to assure they maintain currency in ecosystems, programming technologies, tools, and architecture-related assurance issues. This is related, also, to the expertise issue elaborated in Chapter 2. Competent software architects are rare. They cannot be created through short-term training programs but must be grown through extensive experience with associated ecosystems. Further, successful architects must be involved in design and implementation so that they do not promote obsolete technology and practice. They will exhibit creativity in those things that they specify to ensure a workable architecture emerges. Additionally, they will avoid over-specification where it will inhibit creative opportunities for the developers.

The arm's-length relationship of the DoD with its contractor-based development teams creates challenges in structuring contractual relationships that facilitate the free exchange of information and feedback between developers and other stakeholders—and in supporting the kind of adjustment and refactoring that is required to achieve early validation for crucial architectural commitments.

It may be difficult for some stakeholders to ascertain which kinds of architectural commitments are most essential to success of a particular project. Simple certification of evolving industry-standard ecosystems and APIs, although important to an assurance process, does not necessarily constitute architectural commitment of the kind that is the principal subject of this chapter.

Finally, focus on any architecture should not be viewed as an invitation for the creation of compliance-focused bureaucracy. Competent managers must, on an ongoing basis, assess the choice of architectural constraints for the intended benefits to quality attributes and functional capabilities. Although architects will use standard ecosystems, protocols, and interfaces in defining an architecture, these choices are not equivalent to selection of standards.

In developing a product-line strategy, the PEOs need to consider the fact that the know-how associated with the architecture, its constituent ecosystems, and the diverse related technical resources and tools are all critical factors in a selection. They must also factor into the strategy the cost of tool development and training. Alternatively, they could provide financial incentives for contractors to develop those capabilities independently.

Recommendation 3-2: This committee reiterates the past Defense Science Board recommendations that the DoD follow an architecture-driven acquisition strategy, and, where appropriate, use the software architecture as the basis for a product-line approach and for larger-scale systems potentially involving multiple lead contractors.

Recommendation 3-3: The DoD should enhance existing practices to afford better distinctions between precedented portions of systems and innovative portions of systems, wherein architectures are developed both to encapsulate the innovative elements and to afford maximum opportunity to build on experience and existing ecosystems for precedented elements. These overall architectures, and particularly the innovative elements, should be subject to early and continuous validation, especially in systems that have requirements for interoperation.

The foregoing discussion, findings, and recommendations apply to both precedented and innovative (unprecedented) DoD systems and subsystems. As the DoD considers development of larger systems, including systems of systems, where some of the capabilities are precedented and others are innovative, the acquisition challenges become more difficult. In those cases, the development of software

architecture will be even more significant, in some cases requiring architectural innovation. Indeed, in many cases of large-scale interconnected systems (variously called ultra-scale systems, systems of systems, net-centric systems, etc.), architectural considerations may even be a driver of choices regarding system functionality. That is, once the "scope" of functional capability is identified, then architectural commitments may need to be contemplated before details are worked out regarding those functional capabilities. In this respect, quality attributes or "non-functional" requirements (reliability, scalability, security, performance, etc.) and anticipated interoperation requirements both dominate, because, from a technological perspective, their potential for fulfillment is most directly predicted by architectural decisions. Further, if the architecture effectively supports a defined "scope" of functional capabilities, then many of the details regarding the particular manifestations of those capabilities can be worked out later in the process—when the engineering (and potentially operational) uncertainties regarding the downstream impacts of particular possible choices are reduced.[43]

Finding 3-5: In systems with innovative functional or quality requirements, benefit is derived from an early focus on the most essential architectural commitments and quality attributes, with deferred commitment to specifics of functional characteristics. This approach can reduce the overall uncertainty of the engineering process and yield better outcomes.

These more complex and interconnected systems consist of multiple components interacting, and include functionality that may cut across multiple traditional defense functional areas. Future Combat Systems (FCS) is one such example. This means that building on existing, proven architectures can reduce the amount of innovation required and risk sustained in a project of this sort. Because the power promised by these systems comes at a significant price in complexity (e.g., the multitude of sensors, weapons, and battle command centers), a greater focus is needed on engineering risk when planning the sequence of engineering commitments. In these cases, the biggest lever on engineering risk—and enabler of scale, interoperation, and other critical non-functional requirements—is very often architecture. Moreover, in many of these cases, the overall architecture is a composite of diverse precedented ecosystems structures combined with encapsulated innovative/unprecedented elements whose architecture, in the purely local context, may have associated design risks. In this manner, a massive global uncertainty is replaced by localized uncertainties, which, from a systems engineering perspective, means a consequent reduction in overall systems engineering risk.

With prototyping and instrumentation—the software analogs of the modeling and simulation practices pervasive in the development of physical systems—the engineering risks can be discharged through early validation, thus reducing the overall project risk even for highly innovative projects.

As noted above, architecture is very often the fulcrum of potential for evolution and complexity. For example, how can architectures be developed and validated to support the kind of local autonomy necessary for diverse kinds of vehicles to navigate effectively over mixed terrain? How can software and systems architectures be evolved, for example, as algorithms and machine-learning capabilities improve? Moreover, by specifying interfaces for which testing or measurement is possible, by defining reusable components, and by separating critical from non-critical parts of the system, architecture plays an essential role in localizing uncertainties regarding assurance and thus reducing overall risk related to assurance.

An interesting case study of architecture and evolution in an environment with multiple competing organizations is the architecture of large-scale web systems. This has been a topic of intense interest to startup and established companies for 15 years. One of the key early architectural ideas related to servers was the use of scalable networks of PC workstations and "shared-nothing computation." This

[43] Barry Boehm, Ricardo Valerdi, Eric Honour, 2008, "The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems," *Systems Engineering* 11(3):221-234.

originated in academic research,[44] and in turn was rapidly developed and proven in startup companies such as Inktomi, Google, and others. The academic research approach is quickly evolving into standard "best practice" supported by web development tools and web servers.

Google took several big jumps beyond this architecture to address issues related to much larger scale and the need for new computational abstractions suited to this new category of high-performance data-intensive computations. It shared its ideas through an influential series of papers on the Google File System (GFS)[45] and the MapReduce computational model.[46] The MapReduce model, although seemingly quite radical, builds on ideas from functional programming that have their heritage in the 1960s. The architecture of these systems was replicated and adapted by others including Amazon and Yahoo!. Indeed, Yahoo! and the Apache Software Foundation collaborated to host the open-source Hadoop system, now widely adopted for applications that go well beyond text analysis.[47] Many of the infrastructure-level ideas also fed the development of cloud-computing architectures such as those supported by Amazon, Google, Microsoft, and others.[48] The point of all this is that each player incrementally adapted an existing, mostly proven architecture into a new model, with the result that the end-state (the key computational abstractions and their delivery in modern distributed data centers) appears singularly revolutionary, despite the reality of its evolutionary development.

Recommendation 3-4: The DoD should learn from commercial experience and, in addition, sponsor diverse areas of technical research to help reduce the engineering risk in architecting systems that include unprecedented functional and quality attributes.

Some specific areas of research focus are given in the inventory of topics above, and these are further elaborated in Chapter 5, which offers specific recommendations for addressing the complexity that the DoD will face. A theme that cuts across many of these topics is the idea of developing *modeling and simulation* tools suitable to informing architectural decisions, analogous to the modeling and simulation done for physical elements of many different kinds of DoD systems. This creates the possibility of a try-before-buy approach to key architectural decisions, wherein architectural concepts are modeled using tools, and analysis and testing can be done to assess scalability, performance, robustness, and resiliency to failures and attacks. In other words, the techniques offer a kind of "early validation" whereby engineering risks can be discharged earlier and with lower cost than if the uncertainties persisted until later implementation and test phases of development. By modeling and simulation, the committee means something broader than the current theory and practice of testing and analysis of software code, which focus on conformance of program behavior to specified behavior. The goal is to augment this with tests and analyses that provide information at the earliest possible stages of the process to support evaluation and validation of architecture concepts, interface and framework definitions, and other architectural elements.

A second theme, also noted in the topic inventory, is to develop audit and instrumentation tools to provide early data once architectures are designed and initially populated.

---

[44] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier, 1997, "Cluster-Based Scalable Network Services, *Proceedings of SOSP '97*, St. Malo, France, October 1997. Available online at http://www.cs.berkeley.edu/~brewer/papers/TACC-sosp.pdf. Last accessed August 20, 2010.

[45] For more information, see Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, 2003, "The Google File System," *ACM SIGOPS Operating Systems Review* 37(5):29-43. Available online at http://portal.acm.org/citation.cfm?id=1165389.945450&coll=GUIDE&dl=GUIDE&CFID=90089376&CFTOKEN=82606234. Last accessed August 20, 2010.

[46] For more information, see Jeffrey Dean and Sanjay Ghemawat, 2008, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM* 51(1):107-113.

[47] For more information, see http://hadoop.apache.org/. Last accessed August 20, 2010.

[48] Werner Vogels, 2007, "Enterprise Scale and Beyond," presented at Meeting #2 *Advancing Software-Intensive Systems Producibility*, January 17-18, 2007, Washington, DC. A summary of briefers' presentations and the workshop discussion can be found in NRC, 2007, *Summary of a Workshop on Software Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 20, 2010.

A third theme, relating to the idea of precedent and architectural risk management, is to develop and analyze a family of precedented "scalable architectural patterns" that could provide a well-understood infrastructure of building blocks out of which ultra-large-scale architectures could be designed. This could facilitate the use of multiple suppliers at the architectural and component levels. Additionally, if tools are in place that can support more aggressive restructuring, then a more principled approach can be taken to architectural design that includes iterative development, currently very difficult at the architectural level. This could also enable constructive response even to relatively late-breaking news regarding the consequences of early architectural commitments. Some combination of all these approaches will likely be necessary.

4

Adopt a Strategic Approach to Software Assurance

SOFTWARE ASSURANCE AND EVIDENCE

One of the great challenges for both defense and civilian systems is software quality assurance. Software assurance encompasses reliability, security, robustness, safety, and other quality-related attributes. Diverse studies suggest that overall software assurance costs account for 30 to 50 percent of total project costs for most software projects.[1] Despite this cost, current approaches to software assurance, primarily testing and inspection, are inadequate to provide the levels of assurance required for many categories of both routine and critical systems.[2]

In major defense systems, the assurance process is heavily complicated by the arm's-length relationship that exists between a contractor development team and government stakeholders. This relation-

[1] In "Software Debugging, Testing, and Verification," *IBM Systems Journal* (41)1, 2002, B. Hailpern and P. Santhanam say, "In a typical commercial development organization, the cost of providing this assurance via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost." In *Estimating Software Costs* (McGraw-Hill, 1998), Capers Jones provides a table relating percentage of defects removed to percentage of development effort devoted to testing, with data points that include 90 to 39 percent, 96 to 48 percent, and 99.9 to 58 percent. In *Software Cost Estimation with COCOMO II* (Prentice Hall, 2000), Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece indicate that the cost of test planning and running tests is typically 20 to 30 percent plus rework due to defects discovered. In *Balancing Agility and Discipline* (Addison-Wesley, 2004), Barry Boehm and Richard Turner provide an analysis of the COCOMO II Architecture and Risk Resolution scale factor, indicating that the increase in rework due to poor architecture and risk resolution is roughly 18 percent for typical 10-KSLOC (KSLOC stands for thousand software lines of code) projects and roughly 91 percent for typical 10,000-KSLOC projects. (COCOMO II, or constructive cost model II, is a software cost, effort, and schedule estimation model.) This analysis suggests that improvements are needed in up-front areas as well as in testing and supporting the importance of architecture research, especially for ultra-large systems.

[2] The challenges relating to assurance were highlighted by several briefers to the committee. In addition, this issue is a core concern in the Defense Science Board (DSB), September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Undersecretary of Defense for Acquisition, Technology, and Logistics, at pp. 30-38. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. The 2007 NRC report *Software for Dependable Systems* also addressed the issue of testing and noted, "Testing … will not in general suffice, because even the largest test suites typically used will not exercise enough paths to provide evidence that the software is correct nor will it have sufficient statistical significance for the levels of confidence usually desired" (p. 13). See NRC, Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds. 2007, *Software for Dependable Systems,* National Academies Press, Washington, DC. Available online at http://www.nap.edu/catalog.php?record_id=11923. Last accessed August 20, 2010.

ship—in which sometimes even minor changes to up-front commitments may necessitate amendments to contracts and adjustments in costing—can create barriers to the effective and timely sharing of information that can assist the customer in efficiently reaching accurate assurance judgments. Additionally, it can be difficult to create incentives for the appropriate use of preventive measures such as those referenced in this chapter.

In this chapter the committee first considers the trends related to the challenges of software assurance. It then offers a concise conceptual framework for certain software assurance issues. Finally, it identifies significant technical opportunities and potential future challenges to improving our ability to provide assurance. (Some of these are elaborated in Chapter 5.)

Failures in software assurance can be of particularly high consequence for defense systems because of their roles in protecting human lives, in warfighting, in safeguarding national assets, and in other pivotal roles. The probability of failure can also be high, due to the frequent combination of scale, innovative character, and diversity of sourcing in defense systems. Unless exceptional attention is devoted to assurance, a high level of risk derives from this combination of high consequence and high likelihood.

Assurance considerations also relate to progress tracking, as discussed in Chapter 2—assessment of readiness for operational evaluation and release is based not just on progress in building a system, but also on progress in achieving developmental assurance. Additionally, the technologies and practices used to achieve assurance may also contribute useful metrics to guide process decision making.

Assurance Is a Judgment

Software assurance is a human judgment of fitness for use. In practice, assurance judgments are based on application of a broad range of techniques that include both preventive and evaluative methods and that are applied throughout a software engineering process. Indeed, for modern systems, and not just critical systems, the design of a software process is driven not only by issues related to engineering risk and uncertainty, but also, in a fundamental way, by quality considerations.[3] These, in turn, are driven by systems risks—hazards—as described in Chapter 2 and also in Box 4.1 (cybersecurity).

An important reality of defense software assurance is the need to achieve safety—that is, in war, there are individual engagements where lives are at stake and where software is the deciding factor in the outcome. In many life and death situations, optimum performance may not be the proper overriding assurance criterion, but rather the "minimization of maximum regret." This is exacerbated by the fact that, while a full-scale operational test of many capabilities may not be feasible, assurance must nonetheless be achieved. This applies, for example, to certain systems that support strategic defense and disaster mitigation. The committee notes, however, that there are great benefits in architecting systems and structuring requirements such that many capabilities of systems that would otherwise be rarely used only for "emergencies" are also used in an ongoing mode for more routine operations. This creates benefits from operational feedback and user familiarity. It also permits iterative development and deployment, such as is familiar to users of many evolving commercial online services.

Another reality of defense software that affects assurance is that it is developed by contractors working at arm's length from the DoD. This means, for example, that the information sharing necessary to assessing and achieving assurance must be negotiated explicitly.

There are many well-publicized examples of major defense systems exhibiting operational failures of various kinds that are, evidently, consequences of inadequate assurance practices. A recent example of this type of top-level systems engineering issue was the failure of an F-22 flight management system when it was flown across the international dateline for the first time en route from Hawaii to Japan. In a CNN interview, Maj. Gen. Don Sheppard (ret.) said, "At the international date line, whoops, all systems dumped and when I say all systems, I mean all systems, their navigation, part of their communications,

[3] Michael Howard and Steve Lipner, 2006, *The Security Development Lifecycle,* Redmond, WA: Microsoft Press. See also Box 2.3.

BOX 4.1
Assurance and Cybersecurity—A Brief Consideration

Cybersecurity

Although it is not a principal focus of this report, cybersecurity is an unavoidable and critical dimension of software assurance. It is rarely possible to contemplate software assurance without also giving major attention to security considerations. This is particularly challenging because security, like assurance, must be addressed at every phase of development and the software lifecycle overall.[1]

A system can only be assured if it is well understood. The main text elaborates the concept of a *chain of evidence*, which documents this understanding as traceability from intentions to outcomes, including functional requirements, quality attributes, and architectural constraints. Security adds the additional dimension of threats and attacks. For software, these can occur not only during operations, but also at every stage of the lifecycle, from development through to ongoing evolution and update during operations. The most crude categorization of threats yields three different avenues of attack: (1) external attackers—adversaries gaining access from points external to the system, typically via network connections, (2) operational insiders—adversaries gaining access to a DoD software system through inappropriate privileging, compromised physical access, or compromised personnel, and (3) engineering insiders—adversaries influencing or participating in the engineering process at some point in the supply chain for an overall system. Attacks can have different goals, typically characterized as "CIA"—breaching **C**onfidentiality of data, damaging the **I**ntegrity of data, and disrupting **A**vailability of a computational service. The analysis of possible threats and attacks is a key element of secure software development. This analysis is strongly analogous to hazard analysis (as discussed elsewhere in this report), and it can lead to a host of security considerations to address in the development of systems, for example, relating to identity and attribution, network situational awareness, secure mobility, policy models and usability, forensics, etc. From the standpoint of secure software development, the committee highlights two principal policy considerations, chosen because they are most likely to significantly influence both software architecture and development practice. The first of these relates to *separation*—minimizing and managing the coupling among components in a way that reduces the overall extent of those most sensitive components in a system that require the highest levels of assurance as well as the "attack surface" of those components with respect to the various avenues of attack noted above. The second relates to *configuration integrity*—the assurance that any deviations or dynamic alterations to an operational system are consistent with architectural intent.

Separation

The first example of a security-related chain is the separation chain. Construction of this chain of evidence entails documenting relationships among critical shared resources and the software and system components that should, or should not, have access to or otherwise influence those resources.[2] This chain documents the means by which access to resources is provided—or denied—to the components of a software system that need to rely on those resources. A less trusted component, for example, may be excluded by policy from observing, changing, or influencing access by others to a critical resource such as a private key.

The ability to construct chains of this kind is determined by architectural decisions and implementation practices. Concepts from security architecture such as process separation, isolation, encapsulation, and secure communication architecture determine whether this kind of chain can be feasibly constructed, with minimal exposure of the most sensitive portions of a system. For example, modern commercial PC operating systems are designed to achieve security goals while offering tremendous generality and power

[1] Steve Lipner and Michael Howard, 2006, *The Security Development Lifecycle*, Redmond, WA: Microsoft Press. See also Gary McGraw, 2006, *Software Security: Building Security In*, Boston: Addison-Wesley.

[2] This documentation should be formal wherever possible, such as might be derived from code analysis, verification, and modeling.

in their underlying services and resource-management capabilities. Operating systems more focused on media delivery may offer less generality and flexibility, but may do better in providing assurance relating to security because architectures are designed to more tightly regulate access to resources.

Research advances can expand architectural options for which assurance of this kind can be achieved. This is influenced both through enhancement of architectural sophistication and through the ability to model and assure policies.

Configuration

The second example of a security-related chain is the configuration chain. This chain documents the configuration integrity that is established when a system starts up and that is sustained through operations. The chain, in this case, typically links a known hardware configuration with the full complexity of an overall running system, including software code, firmware, and hardware operating within that configuration. Loss of integrity can occur, for example, when malware arrives over a network and embeds itself within a system. It should be clear that this chain (like the other chain) is significant not only for networked systems but also for any system with a diverse supply chain, due to the differing trust levels conferred on system components. The assurance enabled by this chain is that the assumptions that underlie the construction of other kinds of chains (and the architectural, functional, and other decisions that enable that construction) are reflected in the reality of the code that executes—and so the conclusions can be trusted. Put simply, this chain assures an absence of tampering. This has proven to be a singular challenge for commercial operating systems, as evidenced by the difficulty of detecting and eradicating rootkits, for example.

Documentation of this second kind of chain is complicated by a diversity of factors. One is the dynamism of modern architectures, which afford the flexibility and convenience of dynamically loading software components such as device drivers and libraries. Another is the layered and modular structure that is the usual result of considerations related to development of the second kind of chain. A third factor is assuring configuration integrity of the hardware itself. Including hardware in the chain can be much more challenging than the analogous process for software, because of the added need to "reverse engineer" physical hardware.[3] A fourth factor is derived from the "bootstrap" process through which initial software configurations are loaded onto bare hardware, generally layer by layer. This affords the opportunity of an iterative and ongoing process of loading and integrity checking, such as has been envisioned in the development of the TPM chips that are present on the motherboards of most PCs and game platforms.[4] In this model, the intent is to assure integrity through fingerprinting and monitoring the integrity of software components as they are loaded and configured both through the bootstrap process and during operations. These four factors, combined with a highly competitive environment that discourages compromise on systems functionality and performance, have proven highly challenging for DoD in adopting commercial off-the-shelf operating systems, for example.[5]

A Note on Secrecy

Security-related faults lead to hazards just when attackers are able to exploit those faults to create errors and failures. It may be tempting, therefore, to think that full secrecy of the software code base would preclude such possibilities. For defense systems there are many good reasons for secrecy, but, from the perspective of exploitation of vulnerabilities, over-reliance on secrecy ("security through obscurity") is a

[3] DSB, February 2005, *Report of the Defense Science Board Task Force on High Performance Microchip Supply*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[4] See http://www.trustedcomputinggroup.org/.

[5] DSB, September 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

BOX 4.1 Continued

dangerous approach. There are two reasons. First, faults can very often be detected through sophisticated "black-box" methods, in which attackers probe and poke a system based on hypotheses regarding its likely structure and function—these methods are analogous to those used in software development for operational and systems-level testing. Second, if secrecy enables developers to become complacent about fundamentals, such as appropriate security architectures (see below), then the overall risk can increase dramatically. A minor coding flaw may expose a vulnerability, but with good development and assurance practice that flaw can be readily eliminated either directly through analysis or indirectly through multi-layer defense. An architectural flaw, on the other hand, may be much more difficult or even impossible to mitigate without taking the entire system offline and undertaking significant reengineering.

Opaque Code

As noted in the main text, modern software systems often consist of components drawn from diverse sources. The gradients of trust among components are often complicated by the fact that many components are relatively opaque compared with others—for example, only executable code is available. There are several reasons for this opacity in DoD systems, many of which are driven by commercial considerations related to the protection of intellectual property manifest in source code and design documentation. These considerations may apply both to commercial vendors and to subcontractors who may be potential competitors with their prime contractor on other projects. Indeed, some development organizations may not want to share source code and design information with the government because they are concerned about potential public release or by the possibility of similar requests for access from other governments who they may seek as customers. This is a particular challenge for commercial vendors, who typically conduct business globally and so may face similar requirements from other governments. This risk of exposure may even deter some firms from conducting business in the U.S. government supply chain.

This issue motivates technologies related to sandboxing and isolation, such as those used in web browsers for JavaScript and (as a research goal) technologies for "evidence-carrying code," where evidence of security or safety can be provided in a way that may nonetheless cloak vendor trade secrets.

These considerations notwithstanding, a principal consideration in assurance is the reduction in the extent of code that, in the end, remains opaque to DoD acceptance evaluators. One mechanism, embodied in the Common Criteria model, is the use of mutually trusted third parties to support assurance activities. A key issue is how that evaluation can be done such that two goals are addressed: (1) There is minimal added cost and delay, and (2) Evidence can be produced that protects the interests of the developers and that manifests the necessary links in the various required chains of evidence. The first of these goals could be supported, for example, through the involvement of evaluation teams throughout development. But it could also be addressed through a consistent practice of "evidence production," whereby developers create links in the necessary chains of evidence that can support a more efficient third-party or government evaluation.

One of the challenges in evidence production is achieving a return-on-investment model for evidence production that has the characteristic of "early gratification" for development teams. This was considered unachievable for many years. But there is now evidence in modern team practice, with intensive use of tools for team coordination, defect/issue tracking, and software assurance (unit testing and analysis), that costly after-the-fact practices are giving way to ongoing evidence production, in the same spirit as test-driven development. The second of these goals suggests a number of challenging research problems related to the production of evidence that supports assurance but may also cloak proprietary algorithms from other development teams working on the same system. Both goals also suggest a research challenge related to enhancing the scope of specification of APIs to facilitate demonstration of compliance with API rules of the road. This is significant from an architectural standpoint, because it enables development teams to work more independently of each other, given the added certainty regarding API-mediated interactions.

their fuel systems. They were—they could have been in real trouble. They were with their tankers. . . . The [F-22 crews] tried to reset their systems, couldn't get them reset. The tankers brought them back to Hawaii. This could have been real serious. It certainly could have been real serious if the weather had been bad. It turned out OK. It was fixed in 48 hours. It was a computer glitch in the millions of lines of code, somebody made an error in a couple lines of the code and everything goes." The contact with the tankers was visual: "Had they gotten separated from their tankers or had the weather been bad, they had no attitude reference. They had no communications or navigation. They would have turned around and probably could have found the Hawaiian Islands. But if the weather had been bad on approach, there could have been real trouble."[4]

There Are Diverse Quality Attributes and Methods

Software assurance encompasses a wide range of quality attributes. For defense systems, there is particular emphasis on addressing hazards related to security (primarily confidentiality, integrity, and access of service, see Box 4.1), availability and responsiveness (up time and speed of response), safety (life and property), adherence to policy (rules of engagement), and diverse other attributes. There is a very broad range of kinds of failures, errors, and faults that can lead to such hazards (Box 4.2). Software assurance practices must therefore encompass a correspondingly broad range of techniques and practices.

There is a false perception that assurance can be achieved entirely through acceptance evaluation such as that achieved through operational and systems test. Systems test is certainly a necessary step in assuring functional properties and many performance properties. But it is by no means sufficient. Assurance cannot be readily achieved from testing for many kinds of failures related to security, intermittent failures due to non-determinism and concurrency, readiness for likely future evolution and interoperation requirements, readiness for infrastructure upgrades, highly complex state space, and other kinds of failures.

A comprehensive assurance practice requires attention to quality issues throughout the development and operations lifecycle, at virtually every stage of the process and at all links in the supply chain supporting the overall system. The latter point is a consequence of the observation above regarding the fallacy of relying entirely on acceptance evaluation and operational testing. Although the DoD relies extensively on vendor software and undertakes considerable testing of that software, it also implicitly relies on a relationship founded in trust (rather than "verify") to assure many of the quality attributes (listed above) that are not effectively supported through this kind of testing. This issue is explored at length in a report by the Defense Science Board on foreign software in defense systems.[5]

It is now increasingly well understood by software engineers and managers that quality, including security, is not "tested in," but rather must be "built in."[6] But there are great challenges to succeeding in both "building in quality," using preventive methods, and assuring that it is there, using evaluative methods. The nature of the challenge is determined by a combination of factors, including the potential operational hazards, the system requirements, infrastructure choices, and many other factors.

---

[4] "F-22 Squadron Shot Down by the International Date Line," *Defense Industry Daily*, March 1, 2007. Available online at http://www.defenseindustrydaily.com/f22-squadron-shot-down-by-the-international-date-line-03087/. August 10, 2010. There are also numerous public accounts of software failures of diverse kinds and consequences, such as those cited in the Forum on Risks to the Public in Computers and Related Systems, available online at http://www.risks.org.

[5] Defense Science Board (DSB), 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

[6] This is not a comment about test-driven development, which is an excellent way to transform the valuable evaluative practice of testing into a more valuable preventive practice of test-driven development—building test cases and code simultaneously or even writing test cases before code is written. Note here that "test" should be broadly construed, encompassing quality techniques such as inspection, modeling, and analysis. There are benefits to writing code from the outset that more readily support, for example, modeling, sound analysis, and structured inspection.

BOX 4.2
Faults, Errors, Failures, and Hazards

A *failure* is a manifestation of a system that is inconsistent with its functional or quality intent—it fails to perform to specification. A *hazard* is a consequence to an organization or its mission that is the result of a system manifesting a failure. That is, if a system has been placed in a critical role and a failure occurs, then the hazard is the consequence to that role. For example, if an aircraft navigation system delivers incorrect results, the hazard is the potential consequence to the aircraft, its occupants, its owner, and so on, of incorrect navigation. An *error*, like a failure, is a manifestation when a system is running. But an error can be contained entirely within a system, not necessarily leading to failures. For example, some database systems can detect and remediate "local deadlocks" that involve perhaps a pair of threads, and they can do this in a generally transparent manner. Another example is an unexpected exception (such as might be raised when a null pointer is de-referenced) being handled locally within a component or subsystem. More broadly, architectures can be designed to detect errors, including security problems, within individual components and can reconfigure themselves to isolate or otherwise neutralize those errors.[1] Errors, in turn, are enabled by local faults in code. A *fault* is a static flaw in the code at a particular place or region or identifiable set of places. Examples of faults include points in code where integrity tests are not made (leading to robustness errors), where locks are not acquired (leading to potential race conditions), where data is incorrectly interpreted (leading to erroneous output values), where program logic is flawed (leading to incorrect results), and so on.

In systems that include hardware, probabilistic models are used to make predictions regarding when errors or failures are likely to occur—for example, to compute mean time to failure or expected lifetimes of components. These models are the core of reliability theory, and they can involve complex relationships of conditional probability (i.e., faults that are more likely in the presence of other faults), coupled probability (e.g., when many faults are made more likely in adverse weather), and other complexities. With software, these probabilistic models are less useful, since the failures are caused by intrinsic design flaws that require implementation changes for correction. Intermittent errors in software are thus "designed into the code" (albeit unintentionally). Repair thus means making changes in the flawed design. For embedded software, where the software includes fault-tolerance roles, hybrid models are often most helpful.

This model helps to highlight the challenges associated with effective software testing, inspection, and

[1]Of course, it is possible that the mechanism by which errors are contained results in a loss of information regarding both the errors and the fact that they were contained. This information loss can create dangerous situations. The well known case of the Therac 25 failures (Nancy G. Leveson and Clark S. Turner, 1993, "An Investigation of the Therac-25 Accidents," *IEEE Computer* 26(7):18-41) is a particularly compelling example of the consequences of inadequate information regarding actual error containment in operations. In this case, engineers acted on a false supposition regarding the extent of error containment by a hardware mechanism in operations, resulting in fatal x-ray doses being administered to cancer patients.

Underlying both preventive and evaluative methods are the two most critical broad influences on quality: judicious choice of process and practices, and the capability and training of the people involved in the process. Process and practices can include techniques for measurement and feedback in process execution in support of iteration, progress and earned value tracking, and engineering-risk management. Indeed, a key feature of process design is the concept of feedback loops specifically creating opportuni-

static analysis. The results of tests that fail are manifestations of errors (unit tests) or failures (system tests). Assuming the tests are valid, the engineer must then ascertain which faults may have led to the error or failure manifestations. This reverse-engineering puzzle can be challenging, or not, depending on the scope of the tests and the complexity of the code. Failures in system tests, for example, can derive from the full scope of the code, including incorporated vendor components and infrastructure. Test results are generally of moderate to high value, because they reflect the priorities implicit in the test coverage strategy that guided their creation.[2]

One of the pitfalls of late testing, as would be the case if unit testing were deferred, is that the faults identified may have become very expensive to repair, adding substantially to engineering risk. If the fault is fundamental to the design of a particular interface, then all clients and suppliers that share that interface may be affected as part of the repair process. If the fault is architectural, the costs may be greater, and there may be new engineering risks associated with exploration of alternative options. This suggests both that testing be done at the component level early in the process and that commitments related to architecture and interface design be evaluated through modeling, simulation, and analysis as early as possible in the lifecycle.

The results of inspections, on the other hand, generally point to specific places in code or in models where there are problems of one kind of another. This, from a purely theoretical basis, may be why inspections are sometimes measured as being more effective than testing by a measure of defects found per hour. Because inspections usually combine explicit targeting of issues and opportunistic exploration, the issues found are generally high value.

Static analysis results, including both sound analyses and heuristic analyses, generally point to faults in the code. They thus share with inspections the productivity advantage of avoiding the puzzle-solving inherent in the handling of adverse test results. Additionally, static analysis results can highlight low-probability intermittent errors that might routinely crash continuously operating servers but not be readily detectable using conventional testing. Unlike validated tests, analysis results can include *false positives*, which are indications of possible faults when there are actually no faults. (Unvalidated tests can also produce false positives in cases where the code is "correct," but the test case is not.) Sound static analysis (i.e., static analysis with no false negatives) is used in compiler type checkers and some free-standing analysis tools. Its results are usually tightly targeted to very particular attributes and can lead fairly directly to repairs. Heuristic static analysis results, such as from open-source tools PMD and FindBugs, have considerably broader coverage than targeted sound analysis. But the results are typically less exact, and include false negatives (faults not found) as well as false positives. Additionally, there can be large numbers of results ranging from serious issues to code layout style suggestions. This necessitates an explicit process to set priorities among the results. An analysis of the open-source Hadoop system, for example, can yield more than 10,000 findings.

[2] Test coverage metrics can be useful, but there are many kinds of coverage criteria. Pure "statement coverage" may be misleading, because it may indicate a prevalence of regression tests crafted in response to defects rather than of tests motivated by more "proactive" criteria.

ties for feedback at low cost and with high benefit in terms of reducing engineering risk.[7] Practices can also include approaches to defect tracking, root cause analysis, and so on.

There is overlap between preventive and evaluative methods because evaluative methods are most effective when applied throughout a development process and not just as part of a systems-level acceptance evaluation activity. When used at the earliest stages in the process, evaluative methods shorten

[7] These feedback loops may be conceptualized as "OODA loops"—Observe, Orient, Decide, Act. The OODA model for operational processes was articulated by COL John Boyd, USAF, and is widely used as a conceptual framework for iterative planning and replanning processes.

BOX 4.3
Examples of Preventive and Evaluative Methods

Below are several illustrative examples of *preventive* methods. Underlying all of these particular methods is an emphasis on preventing the introduction of defects or finding them as soon as possible after they are introduced.

- *Requirements analysis*. Assess operational hazards derived from context of use, adjusting operational plans to the extent possible to minimize potential hazard. Assess goals and limits with respect to quality attributes.
- *Architecture design*. Adopt structural approaches that enhance reliability, robustness, and security while also providing flexibility in areas of anticipated change.
- *Ecosystem choice*. Affiliate with ecosystems based on quality assessments of components and infrastructure derived from the associated supply chain.
- *Detail design*. Adopt software structures and patterns that enhance localization of data and control over access.
- *Specification and documentation*. Capture explicit formal and informal representations of functional and quality-attribute requirements, architecture description, detail design commitments, rationale, etc.
- *Modeling and simulation*. Many software projects fail because the consequences of early decisions are not understood until late in the process, when the costs of revising those decisions appear to be prohibitively high, leading to costly workarounds and acceptance of additional engineering risk. It may be perceived by project managers that evaluation cannot be done before code is written and can be run. In fact, a range of techniques related to modeling and simulation can be employed to achieve "early validation" of critical up-front decisions. These techniques include prototyping, architectural simulation, model checking of specifications, and other kinds of analysis.[1]
- *Coding*. Adopt secure coding practices and more transparent structured coding styles that facilitate the various evaluative methods.
- *Programming language*. Select languages that provide first-class encapsulation and controlled storage management.
- *Tooling*. Support traceability and logging structures in tooling, providing direct (and ideally semantics-based) interlinking among related design artifacts such as architecture and design specifications, source code, functional specifications, quality-attribute specifications, test cases, etc.

[1] Daniel Jackson's Alloy model checker is an example of early validation technique for specifications. Daniel Jackson and Martin Rinard, 2000, "Software Analysis: A Roadmap," in *The Future of Software Engineering*, Anthony Finkelstein, ed., New York: ACM, pp. 215-224.

feedback loops and guide development choices, thus lessening engineering risk. (To illustrate the range of methods and interventions related to quality in software, a summary is presented in Box 4.3.)

Judgments Are Based on Chains of Evidence

The goal of assurance methods is to create connections, a set of "chains of evidence" that ultimately connect the code that executes with architectural, functional, and quality requirements. The creation of these chains is necessarily an incremental process, with "links" in the chains being created and adapted as the development process proceeds. An example of a link is a test case that connects code with a particular expectation regarding behavior at an internal software interface. Another link, perhaps cre-

Here are several illustrative examples of *evaluative* methods. These are applied throughout a lifecycle to assess various kinds of software artifacts.

- *Inspection* of the full range of software-related artifacts, ranging from models and simulation results supporting requirements and architecture design to detailed design specifications, code, and test cases.
- *Testing* of code with respect to function, performance, usability, integration, and other characteristics. Test cases can be developed to operate at the system level, for example, simulating web-browser clients in testing e-commerce or other web services systems, or they can operate on code "units" across software interfaces to test aspects of component behavior. Test cases are selected according to a combination of coverage strategies determined by architecture and ecosystem, software design, programming language choice, potential operational risks, secure coding practices, and other considerations.
- *Direct analysis* of source, intermediate, or binary code, using sound tools that target particular quality attributes and heuristic tools that address a broader range of quality attributes.
- *Monitoring* of operational code and dynamic analysis of running code, focused on particular quality attributes. As with testing, monitoring can operate at the system level, including logging and event capture, as well as at the unit level, such as for transaction and other internally focused event logs. Monitoring supports prevention, evaluation, and also forensics after failures occur. Infrastructure for monitoring can support a range from real-time to short-time delayed to forensic analyses of the collected event data. In the absence of other feedback loops, this can assist in focusing attention on making repairs and doing rework.
- *Verification* of code against specifications. A number of formal "positive verification" capabilities have become practical in recent years for two reasons: First, scalability and usability are more readily achievable when verification is targeted to particular quality attributes.[2] Second, new techniques are emerging, based on model checking or sound analysis that support this more targeted verification without excessive requirements for writing formal specifications and assertions in code.

Various process models have been proposed that provide a framework within which these various preventive and evaluative methods can be applied in a systematic fashion, structured, as it were, within Observe-Orient-Decide-Act (OODA) loops of various durations. Two of the most prominent are the Lipner-Howard method (the SDC, or Secure Development Lifecycle) and the method proposed by McGraw.

[2] An example is the Microsoft Static Driver Verifier tool developed by Tom Ball for verifying protocol compliance of Windows device driver code using model checking. See Steve Lipner and Michael Howard, 2006, *The Security Development Lifecycle: A Process for Developing Demonstrably More Secure Software*, Redmond, WA: Microsoft Press.

ated using model-based analysis techniques, would connect this specific interface expectation with a more global architectural property. Another link is the connection of a fragmentary program annotation ("not null") with the code it decorates. A further link would connect that global architectural property with a required system-level quality attribute. Validation of this small chain of links could come from system-level testing or monitoring that provides evidence to support presence of the system-level quality attribute.

This metaphor is useful in highlighting several significant features that influence assurance practice and the cost and potential to achieve high levels of assurance. Here are some examples of influences on the success of assurance practice:

- There is a great diversity of the particular kinds of attributes that are to be assured. These range from functional behavior, performance, and availability, to security, usability, and interface compliance for service application programming interface APIs and frameworks. The Mitre Corporation maintains a catalog, the Common Weakness Enumeration (CWE)[8] that illustrates the diversity in its identification of more than 800 specific kinds of "software weaknesses."
- There is also a great diversity of kinds of artifacts that must be linked in the chains. These include code, design models, architectural models, and specifications of functional and quality requirements. These also include more focused artifacts such as individual test cases, inspection results, analysis results, annotations and comments in code, and performance test results.
- There is a range of formality among these artifacts—some have precise structure and meaning, and others are informal descriptions in natural language or presented as diagrams. (This issue is elaborated below.)
- Components and services encompassed in a system may have diverse sources, with varying degrees of access to the artifacts and support/cooperation in an overall assurance process. Identification and evaluation of sources in an overall supply chain is a significant issue for cybersecurity (see Box 4.1), for which both provenance (trust) and direct evidence (verification) are considerations that influence the cost and effectiveness of an assurance process.
- Many different kinds of techniques must be employed to assess consistency among artifacts and to build links in the chain. The most widely used are testing and inspection. Other techniques that are increasing in importance include modeling and simulation (e.g., for potential architecture choices), static analysis, formal verification and model checking (for code, designs, specifications, and models), and dynamic analysis and monitoring (for code, design, and models).
- Some techniques are based not on reasoning about an artifact or component, but on safely containing it to insulate system data and control flow from adverse actions of the component. Techniques include sandboxing, process separation, virtual machines, etc.[9]
- Different links in the chain may have different levels of "confidence," with some providing (contingent) verification results and others providing a more probabilistic outcomes that may (or may not) increase confidence in consistency among artifacts. Test coverage analysis, for example, can be used to assess the appropriate degree to which a particular set of test results may be generalized to give confidence with respect to some broad assurance criterion.
- Methods or their implementations may be flawed or implemented in a heuristic way that may lead to false positives and/or false negatives in the process of building chains.

Perhaps most importantly, the cost-effectiveness of activities related to software assurance is heavily influenced by particular choices made in development practice—factors that are in the control of developers, managers, and program managers. Here are examples of factors that influence the effectiveness and cost of both preventive and evaluative methods:

- In assurance activities, access is provided not only to source code, but also to specifications, models, and other documentation. Without this information, evaluators must expend resources to "reverse engineer" design intent on code produced within their own organization and create these intermediate models. In the 1980s, a study suggested that, in fact, the DoD spends almost half of its

[8] The CWE inventory (available online at http://cwe.mitre.org/) focuses primarily on security-related attributes. See also, for example, Robert C. Seacord, 2005, *Secure Coding in C and C++*, Boston: Addison-Wesley, for an inventory of potential issues related to not only secure, but also safe and high-quality code. There is substantial overlap of attributes related to safe and quality coding, on the one hand, and security, on the other.

[9] Use of these containment or isolation techniques may create benefits for components that are opaque (some vendor executables, for example) or that are difficult to assure intrinsically (mobile code and scripts in a web services environment, for example). But there are also potential hazards associated with the containment infrastructure itself (such as virtual machine or a web-client sandbox), which must often also be assured to a high level of confidence.

post-deployment cost (47%) reverse engineering its own code.[10] Of course, this reverse engineering was for diverse purposes, but it illustrates the failure of documentation and traceability.

- Traceability exists among the diverse software artifacts including code and the various model and documentation components. The goal, as noted above, is to ultimately connect code with architectural, functional, and quality requirements. In some software engineering groups, evaluators ignore documentation on the premise that it is easier to reverse engineer the code being evaluated (but see above). That is, while the artifacts exist, traceability is lacking, making it difficult both to locate the correct document in a sea of documentation and to verify that the description in the document remains current with as-built code. Modern team-based software tooling has presented a revolution in traceability and logging—for example, each line of code in modern tool-enhanced code bases can have direct links to its complete history including which developers have "touched" that line of code and for what purpose.
- Choices are made regarding architecture, design, and coding that facilitate more definitive evaluation outcomes. These choices relate to formality, explicit complexity in structure, and information hiding and modularity, as well as to the characteristics of possible executions of the code. For example, distributed and concurrent systems can, for an unchanging input, exhibit different behaviors with each run. This is due to the asynchrony often characteristic of concurrent execution. When errors are unlikely but possible, testing and even inspection may not offer sufficiently useful results.[11]
- Product-line and ecosystems choices can enable leveraging of assurance activity across multiple projects. This benefit is proportional, however, to the extent that assurance techniques can be composed, which in turn is enabled by our ability to model assurance-related attributes at component or protocol interfaces. (This is a research issue identified in Chapter 5.)
- Choice of programming language (and coding style) can significantly influence ability to assure. Highly complex "tangled code" in a language such as C (which lacks first-class encapsulation and controlled access to storage) may present formidable barriers to evaluative methods in achieving confident assurance judgments when compared, for example, with well-structured programs in modern languages such as C#, Java, and Ada that have comparable functionality.[12] In these latter cases, "well-structured" means two things: First, modular structures can be crafted using modern type systems to replace tangled complexity with organization. Second, intrinsic support for information hiding and encapsulated data simplifies the structure of the various links in the chain of evidence that need to be constructed.

All evaluative methods are challenged by the difficulty of defining the scope of the operating environment that may be delineated as the "boundaries" for evaluation.[13] Unanticipated features of the operational environment that affect system operation may influence not only hazard, but also the validity of requirements. An example of such a scoping error occurred during a test of an F-22 that originated at Edwards Air Force Base and flew to an altitude where it became exposed to the many radio emitters in the Los Angeles basin. This was the first such intensive radio exposure in the test process for the jet,

[10] See Girish Parikh and Nicholas Zvegintzov, eds., 1983, *Tutorial on SoftwareMmaintenance*, Silver Spring, MD: IEEE Computer Society Press. See also Center for Software Engineering at the University of Southern California (USC), 2000,"COCOMO II," Los Angeles: University of Southern California. Available online at http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf. Last accessed August 20, 2010.

[11] An early example was the start-up failure in establishing communications among the five computers on the NASA Space Shuttle on April 10, 1981. Later investigation of the design showed that there had been a 1 in 67 chance that the computers would not synchronize. This meant that, in testing, there was a better than 98 percent chance that the error would not be observed. If there had been anticipation of stochastic phenomena, the error could have been found with more pervasive testing. But in practice this is infeasible for two reasons: (1) there are significant errors that might occur with much lower frequencies, and (2) there are too many different kinds of interactions that might prompt this kind of testing.

[12] The performance gap between "lower level" languages such as C and modern encapsulation-based languages has generally been closed and, indeed, modern languages may offer better performance in many cases since runtime checks can be eliminated when static verification is achieved by compilers for typing and encapsulation properties, for example.

[13] Michael Jackson, 1995, *Software Specifications and Requirements: A Lexicon of Practice, Principles and Prejudices*, Boston: Addison-Wesley.

despite much testing. Unexpectedly, the software concluded that the jet was under attack, and it went into an electronic defensive mode. Crew were forced to shut down all functions to prevent unintended consequences. This experience led the F-35 Joint Strike Fighter developers to test all their software in a fully realized flying testbed well before the actual fighter was flown. This flying testbed is now one of many steps in a highly comprehensive (and expensive) process of operational testing in support of acceptance.

Practices Influence Feasibility and Cost of Assurance

The examples above illustrate that development practices and technologies can profoundly influence the ability to achieve successful and cost-effective evaluation outcomes. These development choices range from architectural choices to choices of programming language and coding style. As noted above, complex tangled code is more difficult to evaluate than structurally simpler code, regardless of whether the evaluation is done using testing, inspection, static analysis, or model checking. It may be, for example, that the 2 percent performance improvement that is created through the additional complexity may not be justified when the added evaluation costs are considered.[14]

One of the great benefits of modern tooling is that a much more comprehensive record of development can be created to facilitate evaluation. When more of the various development-related artifacts are formal (i.e., have precise structure and meanings), then tooling can be used to greater advantage in both prevention and evaluation (as well as in prototyping and other analogs of the modeling and simulation common in the development of physical systems). Degree of formality is an important characteristic of software-related artifacts, discussed at greater length below.

Finding 4-1: The feasibility of achieving high assurance for a particular system is strongly influenced by early engineering choices, particularly architectural and tooling choices.

Assurance Techniques and Results Can Benefit Developers Directly

Because of recent advances in traceability, evaluative techniques, and expressiveness of models, this record of artifacts associated with development is gaining considerable value in contributing to the creation of chains of evidence. When development teams see immediate benefits from the evidentiary material, they are naturally led to adopt a broader range of preventive practices to create additional links in the chain of evidence. It is increasingly apparent that modern assurance techniques can provide immediate benefits in the form of direct feedback loops and greater transparency in the processes implemented by small teams and even by individual developers. The techniques and associated models are also enablers of flexibility and evolution, which are essential in long-lived software systems of all kinds, because of the rapid changes in operational requirements, infrastructure, ecosystems, and underlying hardware capabilities.

SOFTWARE ASSURANCE FUNDAMENTALS

Software Reliability Is Different

Unlike other engineering materials, software does not wear out or suffer transient faults. But it can suffer transient *errors*, for example, because of concurrency (see Box 4.2). This is both an obvious and

[14] Even if choices related to architecture and language affect performance or code size by observable constant factors, there is a pareto principle that suggests that this can be mitigated through performance-focused tuning of a small number of "hot spots" in code. This enables the benefits of superior structure to be realized without adverse performance cost. This point notwithstanding, the idea of a tradeoff of speed against structure and safety is not necessarily principled and may, in the long haul, be incorrect.

subtle point. It is obvious in the sense that there is no analog of metal fatigue, rust and oxidation, or other kinds of physical deterioration or environmentally induced change in physical properties. It is subtle because software is often the mechanism of choice for handling such faults in associated hardware, such as sensors and actuators in a robotic or cyber-physical system, or faults in underlying computing hardware such as processor chips, memory, and communication channels. When software delivers "bad results," including transient errors, these are due to "permanently faulty" software design, which is addressed by changes in the software code—that is, a "new software design" in the sense of changing the mechanism that is implemented.

Despite these differences, the terminology of reliability is usefully applied to software.[15] The core of the terminology is four words: fault, error, failure, and hazard. These are defined and illustrated in Box 4.2.

Information Loss and Traceability

As noted above, the software engineering process is almost always characterized by cycles of information loss and recovery. Although code[16] is all that is necessary for the software to operate, considerable additional information is needed to effectively support ongoing evolution of the software over its lifespan. Some of this information is formal—that is, its expressions are precisely structured and have exact meanings—while other information is "informal," which typically means expressed in the form of natural language text, presentation charts, sketches, and informal diagrams. Examples of formal information are test cases, assertions in code, certain kinds of design models such as unified modeling language (UML) StateCharts and formal structural architectural models (such as Acme). Examples of informal information include comments in code, design description and rationale, structured API documentation (such as Javadoc), and architecture and design diagrams such as from UML.

Two small scenarios illustrate the value of this kind of information in the design process:

1. A planning process for system enhancement leads to reconsideration of a principal architectural commitment such as choice of ecosystem, design of structural architecture, or choice of infrastructure components. Original designers and developers are sought out to help a new team of planners to understand elements of decision rationale for the as-built system, including other alternatives that were considered and why those choices were made.
2. An internal algorithmic enhancement is made in a module that connects to the rest of the system through a software interface or a network protocol. Questions arise concerning particular "rules of the road" for that interface or protocol, and they can be resolved only through an examination of other modules in the system that operate through that interface or protocol. Other questions arise due to the possible dependency of client code on "accidental features" visible through an interface or protocol but not intended to be promised.

Software producibility is directly influenced not only by the *extent* of design-related information that is retained and managed, but also by the means by which this design-related information is represented.[17] There are four dimensions of representation that are most significant. These are formality, modeling, consistency, and usability.

---

[15] Daniel P. Siewiorek and Robert S. Swarz, 1998, *Reliable Computer Systems: Design and Evaluation*, Natick, MA: AK Peters, Ltd.

[16] "Code," in this context, includes both executable files and associated declarative configuration files such as the XML files often used in .Net and Java EE web systems.

[17] The committee uses the phrase "design-related information" in a broad sense to include not only architectural and structural commitments, but also other commitments related to quality and functional attributes not otherwise explicit in the code itself.

Formality

When information is represented formally, tools not only can make maximum use of the meanings that are expressed but also can rely on those meanings as being exact. Tools can also use informal information, of course, but the inexactness of meanings limits their ability—and consequently the ability of software developers—to rely on any particular meaning as being correct. This suggests a strong bias for formality.[18] The extent of formality (i.e., expressiveness of formal notations) is limited, however, by the state of practice regarding what we know how to express. Much of the advancement in programming languages (assembly language to C to C++ to C# and Java, for example) and design notations (informal ad hoc structural diagrams to the UML family, for example) is enabled by advancements in technical progress by the research community. One of the challenges is understanding what scope of the worldly context of operations must be modeled in order to support reasoning regarding the full range of functional and quality requirements.[19] This is an area where there has been steady progress in research, along with significant influence of that research on practice.

Formal information can be very simple, such as references to version numbers, identifiers in defect databases, web links (URLs), or the extensive structured metadata in a defect management database. This illustrates the notion of partial formality, sometimes called "semiformal" or "semi-structured," wherein formal information (such as web links) is embedded in informal texts, or vice versa (e.g., textual comments embedded in code). Another example, in the defect databases, is the fact that there is also considerable latitude for informal expression within the overall structure of the wealth of "formal" metadata—for example the words used to describe the defect or the constituent messages in the "blog" record associated with the defect. Formality can also be semantically very "deep," such as the temporal logic specifications used to express models for model-checking tools.[20]

A key insight is that any step from informal text to structured metadata facilitates traceability and analysis. These steps involve making structure more explicit and identifying precise meanings for the elements of the structure. This is not to say that all models should be formal—achieving formality can create constraints on flexibility and expressiveness. This is why there is so much partial formality. But it also reminds us that incremental steps can be made as research progresses.

Semantic expressiveness is a key distinguishing feature among programming languages, within which small steps can make a considerable difference. For example, the first-class typing of Ada, C#, and Java creates significant advantages for development teams in managing structural aspects of larger-scale systems, and particularly in ongoing assessment of consistency of as-built code with architectural specifications. The C language does not afford such advantages. Although the C++ language gives some of the benefits, it is possible to "bypass" the protection mechanisms in C++ programs and thus lose some of the benefits. Much of the subject of modern programming language research is how to increase the expressiveness of type systems and other structuring mechanisms to facilitate more modular management of large evolving code bases as they evolve and more concise expression of abstractions represented in the code and their relationships.

---

[18] This does not necessarily relate to "formal methods" as traditionally construed. See footnote 20 below. The idea of "formality" is about precision of structure and meaning—and even HTML tags confer a small increment of formality. This is distinct from many of the methodologically focused ideas proposed under the rubric of "formal methods" over the past four decades. Much of the recent success of mathematically based approaches that build on the tradition of formal methods has been in areas often called "lightweight formal methods"—approaches that trade scope and generality for scalability and ease of use. These more scalable approaches include model checking, sound static analysis, and some approaches based on assertion-passing verification. Because they focus more narrowly on particular quality or functional attributes, these approaches have achieved success in professional development practice. An example is Microsoft's use of diverse analysis tools such as SLAM, PreFast, Spec#, and others.

[19] This issue is explored at length in Michael A. Jackson, 2001, *Problem Frames: Analysing and Structuring Software Development Problems,* Boston: Addison-Wesley.

[20] The term "formal methods" refers to techniques for reasoning about code or design models, generally focusing on logical relationships between specifications and the code or models.

Modeling

From the standpoint of assurance, models of all kinds—architecture, design, performance, structural, and semantic—form the intermediate way-points that facilitate linking (in the chain of evidence) of executable code with requirements of various kinds. The way-points include "domain-oriented" models related to requirements.[21] The UML family of design models includes models that are more formal, such as StateCharts, and others that are less formal, such as deployment diagrams. The advantage of the more formal models is that there is more that tools can do to support traceability and analysis. StateCharts has a precise semantics rooted in state machines, which enables creation of a range of tools for analysis, simulation, consistency checking with code, and the like.

There are benefits, of course, when models can not only support the software development process and management of engineering risks (e.g., through simulation and analysis), but also facilitate the activities related to assurance. Many of the topics identified in Chapter 6 relate to modeling and the use of models for various purposes.

Tools such as model checkers and static analysis tools are informed by formal specification fragments, which are a kind of model. These are sometimes expressed in self-contained specifications (e.g., linear temporal logic specifications or Alloy specifications for model checkers) and sometimes use fragmentary annotations associated with code or models. Some verification tools make use of highly expressive specification languages for functional properties.

In general there is an advancing frontier from informal to formal models—actually from less formal to more formal models—and modern tooling is creating momentum to push this frontier more rapidly and effectively. In Chapter 5, there is discussion regarding research goals related to both advancing modeling and specification capability and also to improving techniques and tools for reasoning and analysis. Examples include techniques ranging from theorem proving, model checking, and analysis to type modeling and checking, architectural and design analysis, and analyses related to concurrency and parallelism. Much of the recent progress in program analysis, which is particularly evident in certain leading vendor development practices, is built on these ideas.

Consistency

Information in a software development process is gathered incrementally over time. Almost always, systems are evolving and so are detailed choices regarding architecture, requirements, and design. A seemingly unavoidable consequence is a loss of consistency within the database of information captured over time. Indeed, developers often set aside documents and model descriptions, and resort to interviewing colleagues and doing reverse engineering of code in order to develop confidence in the models they are building or evolving. Precision in models (formality) can be useful in achieving consistency when tools can be used to analyze consistency on an ongoing basis. Tool use ranges from maintenance of batteries of regression tests to the use of verification and analysis tools to compare code with models. With both formal and informal information, explicit hyperlinking can expose interrelationships to developers and enable them to more readily sustain consistency among design artifacts.

Extensive hyperlinking is a feature of modern development tools, including team tools and developer tools. It is an essential feature, for example, of modern open-source development and build environments.[22] With automated tools, a very fine granularity can be achieved without adding to developer effort. For example, an open-source developer can check in code by submitting a simple "patch" file,

[21] Requirements always start with informal articulations that are made precise and potentially formal (in the sense of this chapter) through the development process. One of the great benefits of high-quality models for requirements and associated domain concepts is the opportunity for early validation. These models can include scenarios, use cases, mock-ups, etc.

[22] Linking and other kinds of support for traceability are supported in most commercial development tools and in high-end open-source ecosystems. An example that can be readily explored is the Mozilla development ecosystem—see, for example, code and tools at https://hg.mozilla.org/mozilla-central.

and from this the tools can update the information database in a way that shows the identity of the developer who last changed every individual line of code, along with some informal and semi-formal rationale information such as a reference to a file version number and an identifier from the issue/defect database.

Usability

Even the highest quality information does not add value if it is not readily accessible and applicable by the key stakeholders in the software development process—developers, managers, evaluators, and others. With respect to search, for example, there are enormous differences in efficiency between traditional paper documents and electronic records. Augmenting search with linking and with direct support for anticipated workflows is another large step in efficiency. Choice of representation for expressing design information and models can also make a significant difference—"developer-accessible" notations can reduce training requirements and lower barriers to entry for developers to capture information that otherwise might not be expressed at all.

Indeed, we can contemplate a concept of "developer economics" that can be used as a guide for assessing potential motivation of individual developers in using assurance-related tools. An example of bad developer economics is when a developer or team is asked to devote considerable time and effort to expressing design information when payback is uncertain, diffuse, or most likely far in the future. A goal in formulating incentive models that motivate developer effort (beyond management or contractual mandates) is to afford developers increments of value for increments of time invested in capturing design information, and to provide that value as soon as possible after the effort has been invested. Thus, when a developer writes a single-unit test case, it becomes possible both to execute that test case right away on an existing small unit, and to validate the test case against other design information (and to capture links with that design information to support consistency). This "early gratification incrementality" can be a challenge to achieve for certain kinds of tools and formal documentation, however. Success in achieving this "early gratification" is one of the reasons why unit testing has caught on, and model checking and analysis are also emerging into practice.[23]

Finding 4-2: Assurance is facilitated by advances in diverse aspects of software engineering practice and technology, including modeling, analysis, tools and environments, traceability, programming languages, and process support. Advances focused on simultaneous creation of assurance-related evidence with ongoing development effort have high potential to improve the overall assurance of systems.

CHALLENGES FOR DEFENSE AND SIMILAR COMPLEX SYSTEMS

Hazards

The extent and rigor adopted for an evaluation process is most directly influenced by the potential hazards associated with the intended operational environment. Missile launch control, cryptographic tools, infusion pumps for medication administration, automobile brake systems, and fly-by-wire avionics are all "critical systems" whose design and construction are profoundly influenced by considerations of evaluation and assurance. For many critical systems, standards have been established that regulate various aspects of process, supply-chain decisions, developer training and certification, and evaluation. These standards are ultimately directed toward assurances regarding quality attributes in running code. From the particular perspective of assurance, any focus on aspects other than the intended delivered

[23] Difficulty in achieving this kind of incrementality has been a challenge to the adoption of emerging prototype functional verification systems.

code (and its associated chains of evidence) is intended either as a predictor of ultimate code quality or, often, as a surrogate for direct evaluation of some critical quality of that running code. The latter approach is often used as a "work-around" when direct evaluation is thwarted by the raw complexity of the system or the inadequacy of methods and tools available for direct evaluation.

Indeed, system managers often feel that they face an uncomfortable tradeoff between enhancing the capability of a system and delivering a high level of assurance. This folkloric "quality-capability tradeoff" is particularly challenging because it may be difficult to know exactly where on the quality axis a particular design is likely to reside. Greater incentives for quality have had the effect of "pushing outward" this tradeoff curve for both preventive and evaluative methods. This observation explains, for example, why vendors such as Microsoft have made such a strong commitment to advancing in all areas of prevention and evaluation, because it enables them to offer simultaneous increases in quality and capability.

Capability and Complexity

A major complicating factor in software assurance for defense is the rapid growth in the scale, complexity, and criticality of software in systems of all kinds. (This is elaborated in Chapter 1.) This growth adds to both factors in the risk product, including extent of consequence (hazard, due to the growing criticality of software systems, and cost of repair, due to the growing significance of early commitments) and potential for consequence (due to complexity and interlinking with other systems). The transition to fly-by-wire aircraft, which was for many years loudly debated, is an example of the growing consequence of software. In the commercial world, we are now analogously moving to "drive-by-wire" vehicles, where the connections between brake and accelerator pedals and the respective mechanical actuators are increasingly computer mediated. The benefits are significant, in the form of anti-lock braking, cruise control, fuel economy, gas/electric hybrid designs, and other factors. But so are the risks, as documented in recent cases regarding software upgrades for the brake mechanisms for certain Toyota and Ford vehicles.

An example of the risks of fly-by-wire were demonstrated when an F-22 pilot had to eject from his aircraft (which eventually crashed) when he realized that, due to an unexpected gyro shutdown, he had no ability to control the aircraft from the cockpit. He realized this only after takeoff, when the aircraft initiated a series of uncommanded maneuvers. In modern fighters, if the Vehicle Management System computers (VMS) are lost, so is the aircraft.

As noted in National Research Council reports, more constrained domains such as medical devices and avionics benefit from rigorous standards of quality and practice such as DO-178B.[24] These standards prescribe specific documents, process choices (including iterative models), consistency management and traceability practices, and assurance arguments ("verification") that include various links of the chain, as described earlier in this chapter. These approaches are extremely valuable, but they also appear to be more effective in domains with less diversity and scale than is experienced in DoD critical systems.

Complexity and Supply Chains

An additional complicating factor in software assurance for defense is the changing character of the architecture and supply structure for software systems generally, including defense software systems. The changes, which are enabled by advances in the underlying software technologies, particularly related to languages, tools, and runtime architectures, allow for more complex architectures and richer and more diverse supply chains. Even routine software for infrastructure users such as banks, for example, can involve dozens of major modules from a similar number of vendor and developer

[24] NRC, Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds., 2007, *Software for Dependable Systems*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11923. Accessed August 20, 2010.

organizations, as well as custom software components developed by multiple independent in-house development teams. This is in addition to the defense and government challenges of the customer and key stakeholders working at arm's length from the development teams.

When systems are modular and component-based, there are sometimes opportunities to structure the assurance task in an analogously modular fashion. Unfortunately, many critical software attributes do not "compose" in this fashion, but there are some that do. For example, type correctness of software in modern languages such as Java, C#, and Ada is composable, which permits separate compilation of distinct modules. But without composability, the problem of creating "links" in the assurance chain can rapidly become intractable. Composability is therefore an important goal in the design of models, languages, and analysis capabilities.

Additionally, modern systems make greater use of complex software frameworks and libraries. This is a great success in reuse, but there is also great complexity. Frameworks provide aggregate functionalities such as graphical user interaction, application server capability, web services support, mobile device capabilities, software development environments, enterprise resource planning (ERP), and the like. These frameworks embody many of the technical commitments associated with the ecosystems described in Chapter 1, and they now appear ubiquitously in larger-scale commercial applications. A framework is different from a library, roughly, because it embodies greater architectural commitment, including the structure of its associated subsystems, patterns for the flow of control, and representations for key data structures. This approach, which is enabled by modern object-oriented technology and languages, greatly reduces engineering risk for framework users, because the established frameworks embody proven architectures. But it does create some assurance challenges due to the complexity of the relationships among the framework, its client code, and potentially framework add-ins that augment capability in various ways.

Frameworks and Components

The success of component-based architectures, libraries, and frameworks has led to larger and more capable software applications that draw from a much greater diversity of sources for code. This is a mixed blessing. On the one hand, highly capable and innovative applications can be created largely by selecting ecosystems and assembling components, with a relatively very small proportion of new custom design and code development. Often the overall architecture can be highly innovative, even when it incorporates subsystems and components drawn from established ecosystems. This approach is particularly well suited to incremental methods that facilitate accommodation of the refresh cycles for the various constituent components. It also facilitates prototyping, because functional capabilities can often be approximated through the assembly process, with additional custom code added in later iterations to tailor to more detailed functional needs, as they become better understood.

Trust

This model, while attractive in many respects, poses significant challenges for assurance. Because there are diverse components from diverse sources, there will necessarily be differences in the levels of trust conferred on both components and suppliers. This means that, in the parlance of cybersecurity, there are potential attack surfaces inside as well as outside the software application and that we must support rigorous defense at the interfaces within the application. In other words, the new perimeter is within the application rather than around it or its platform. This can imply, for example, that the kinds of architecture analyses alluded to in Chapter 3 that relate to modularity and coupling may also be useful in assuring that among components in a system (e.g., involving access to data or control of resources) there is no "connectivity" other than that which is intended by the architects.

This new reality for large systems poses great challenges for assurance, because of the potentially reduced ability to influence the many sources in the supply chain and also because of the technical

challenges of composing assessment results for individual components and subsystems into aggregate conclusions that can support an assurance case.

Vendor components are very often accepted on the basis of trust and expectations rather than direct analysis. There are both technical and legal barriers to direct analysis that often thwart the ability of the DoD to make sound assessments that can lead to reliable conclusions regarding assurance. There are several options in these cases. One is to employ a formal third-party assessment process such as Common Criteria (ISO 15408), which is in fact derived from the old "Orange Book" process defined in the early 1980s. These processes can be expensive and can create delay.[25] Additionally, results can be invalidated when components must be configured, plug-ins are added, or other small changes are made such as adding device drivers to an operating system configuration. There has been much consideration of alternate approaches to such assessments. (Detailed consideration of this issue is beyond the scope of this report, but consideration is given in the referenced DSB report.[26])

TWO SCENARIOS FOR SOFTWARE ASSURANCE

To illustrate evaluative techniques and the value of preventive techniques when software is developed at arm's length, the committee presents two speculative scenarios for software assurance. In the first scenario, evaluators are given full access to an already existing software system that is proposed for operational release. The access includes source code for all custom development as well as all associated development documents. The evaluators also have access to threat experts, and they may have the opportunity to interview members of the development team. In the second scenario, a similar system is developed, but evaluators have access to the development team from the outset of the project, and the development team leaders have specific contractual incentives to obtain favorable judgments of high assurance.

The first scenario, which is fully after the fact, may be read as a strawman for the second and more desirable scenario. Unfortunately, an after-the-fact response such as sketched in the first scenario is all too often called for in practice—and indeed in some cases may be optimistic due to the opacity of many code and service components.

First Scenario—After the Fact

In the informal narrative below, the committee starts with the first scenario and then (under the same paragraph headings) explores the potential benefits of the greater access in the second scenario.

- *Hazard and requirements analysis.* The first step for the evaluators is to engage with the threat experts and the operational stakeholders for the purpose of identifying the key hazards. These could include hazards related to quality attributes: security hazards (e.g., confidentiality, integrity, and access in some combination), safety hazards (e.g., related to weapons release), and reliability and performance hazards. This will include identification of the principal hazards relating to functional attributes—correctness of operation, usability and ergonomic considerations, and compliance with interoperation requirements

[25] The Common Criteria standard (ISO 15408) is generally considered to be more successful for well-scoped categories of products such as firewalls and other self-contained devices—as contrasted with general-purpose operating systems, for example. Success with Common Criteria is also challenged by dynamic reconfiguration, such as through dynamically loaded libraries, device driver additions, and reconfiguration of system settings by users and administrators. Additionally, much of the evaluation undertaken through the Common Criteria process is focused on design documents rather than on the code to be executed. There may be no full traceability of executing code corresponding to the evaluated design documents.

[26] DSB, 2007, *Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software*, Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Available online at http://stinet.dtic.mil/oai/oai?&verb=getRecord&metadataPrefix=html&identifier=ADA473661. Last accessed August 20, 2010.

and, more generally, with standards associated with interlinked systems (ultra-scale, net-centric, system of systems, etc).

- *Architecture and component identification.* The system and its associated documents are then analyzed to determine the overall intended and as-built system architectures. The intended architecture may not correspond exactly to the as-built, but it should be as close as possible, with deviations plausibly explainable as design or coding defects. As part of this process, the internal component structure of the system is modeled, including the adoption of off-the-shelf components and frameworks from established ecosystems. For example, if the system uses web capabilities, then there will likely be major subsystems implemented as configured vendor frameworks. The result of this step is an architectural model, an identification of the principal internal interfaces that mediate interactions among components (frameworks, libraries, local services, network-accessed services, custom components, etc.), and an identification of significant semantic invariants regarding shared data, critical process flows, timing and performance constraints, and other significant architectural features.[27]
- *Component-level error and failure modeling.* If successful, the architectural analysis yields an understanding of principal constraints on the components of the system that relate to attributes such as timing, resource usage, data flows and access, user interaction constraints, and potentially many other attributes depending on the kind of system. This process, and also the architecture analysis process, is informed by documents and developer interviews.
- *Supply-chain and development history appraisal.* Based on information regarding component sourcing and supply-chain management practices, levels of trust are assigned to system components. This will inform priority setting in assessment of the individual components. Custom components from less-trusted sources may merit greater attention, for example, than off-the-shelf commercial components from more trusted sources. A similar analysis should apply to services (e.g., cloud services, software-as-a-service capabilities, etc.). Open-source components afford visibility into code, rationale, and history. They may also afford access to test cases, performance analyses, and other pertinent artifacts. It is also helpful, from the standpoint of security threats (see Box 4.1), to assess detailed historical development data. This can include not only data regarding producer/consumer interfaces within the supply chain, but also, when possible, code check-in records from modern development databases (such as captured in open-source systems such as SVN and CVS and similar commercial products and services).
- *Analysis of architecture and component models.* Proceeding on the (as yet unverified) assumption that component implementations are consistent with their constraints, the models at the granularity of architecture and component interactions can be subject to analysis. Because of the diversity of attributes of the models that can trace to the identified failures and hazards, multiple modeling exercises are likely to be undertaken, each focusing on particular attributes. When the models can be rendered formally, then tools for semi-automated analysis can be used for model checking, theorem proving, static analysis (at model level), simulation, and other kinds of mathematically based analysis. If certain models can be formalized only partially or not at all, then a more manual approach must be adopted to undertake the analysis.
- *Identify high-interest components.* Component analyses can be prioritized on the basis of a combination of trust level (from the supply-chain analysis) and potential role with respect to hazards, or "architectural criticality." Greater attention, for example, would be devoted to a component that handles sensitive information and that is custom developed by an unknown or less trusted supplier.
- *Develop a component evaluation plan.* The evaluation plan involves allocating resources, setting priorities, identifying assurance requirements, and establishing progress measures on the basis of the analyses above.
- *Assess individual components.* This can involve a combination of evaluative techniques. "Static"

[27] This documentation, focused on succinct renderings of traceability and technical attributes, should not be confused with the "for the record" documentation often required with development contracts—which may be of limited value in an assurance exercise that relies on efficient tool-assisted evaluation.

techniques, which do not involve executing the code, include inspection (with design documents), sound static analysis, and heuristic static analysis. These analyses may involve the construction of various kinds of abstract models that can themselves be analyzed to assess various functional and quality attributes. This activity is facilitated when models can be made more formal—informally expressed models necessarily require people to make interpretations and assessments. The analyses may also involve "dynamic" techniques, which involve execution of the code, either in situ in the running system (analogous to in vivo testing in life sciences) or in test scaffolds (analogous to in vitro testing in life sciences). If the project had used unit testing, then scaffold code would be included in the corpus, and this could be adapted and reused. Dynamic methods also include dynamic analysis and monitoring and can be used to inform the development of static models to provide assurance in cases where this is significant—particularly concurrent and performance-sensitive code. The results of this assessment are in the form of an identification of areas of confidence and areas of remaining assessment risk with respect to the component interface specifications derived from the architecture analysis.

- *Select courses of action for custom components.* On the basis of the identification of high-interest components and the component assessment results, specific options are identified for mitigation of the remaining assessment risks. These options could range from acceptance of the component (a positive assurance judgment) to wholesale replacement of the component. Intermediate options include, for example, containment ("sandboxing" the component behind a façade that monitors and regulates control and data flows, either within the process or in a separate process or virtual machine), refactoring, and other kinds of rework that might lead to more definitive assessment results. For example, simplification of code control structure and localization of state (data) can greatly facilitate analyses of all kinds. On the other hand, if there are major issues that afflict multiple components and the value is deemed sufficient, then this kind of refactoring and rework could be done at the architectural level, facilitating assessment for multiple components.
- *Select courses of action for opaque components and services.* For opaque components (typically products from vendors), the options are more constrained. In these cases, the extent of the intervention may be influenced by the extent of trust vested in the particular vendor in its supply-chain role. When trust is relatively low, potential interventions include sandboxing (as noted above) and architectural intervention to assure that the untrusted component does not have access to the most sensitive data and control flows. Outsourced services, for example, can also be sandboxed and monitored. An alternative is to replace the component or to rework the arm's-length contractual arrangements to facilitate access and evaluation.
- *Refine system-level assessment.* On the basis of the results of the component assessments and interventions (where appropriate and practical), architecture-level refactoring can sometimes be considered as a means to improve modularity, isolating components for which high levels of assurance cannot be achieved. Most importantly, the architectural-level models should be reconsidered in the light of the information acquired and verified in the foregoing steps. This reconsideration should focus on the hazards, quality attributes, and functional requirements as identified in the initial steps. If the component- and architecture-level assurances do not combine to yield sufficient assurances for the hazards identified, then more drastic options need to be contemplated, including canceling the project, redefining the mission context to reduce the unaddressed hazards, revising initial thresholds regarding system risks, or undertaking a more intensive reengineering process on the offending components of the system and/or its overall architecture. As noted in Chapter 3, reworking architecture commitments at this late stage can be very costly, because there can be considerable consequent rework in many components.

This scenario is intended to illustrate not only the potential challenges in an evaluation process, but also some of the added costs and risks that exist due to insufficiency either of effort in the "preventive" category or of evaluator involvement in the development phase. In the second scenario, the committee briefly considers how these steps might be different were the evaluators and developers to work in partnership during the development process rather than after the fact.

Second Scenario—Preventive Practices

The steps are the same as those for the first scenario, but the descriptions focus on the essential differences with the after-the-fact scenario above. This scenario should make evident the value of incentives in the development process for "design for assurability."

- *Hazard and requirements analysis.* This step is similar, but performed as part of the overall scoping of the system. Because architecture is such a primary driver of quality attributes and assurance (as illustrated above), in this preventive scenario, a savvy manager would couple the architecture definition with the hazard analysis and, if possible, limit early commitment regarding specific functional characteristics to broad definitions of the "scope" of the system (see Chapter 2). At this stage, the first set of overall progress metrics is defined, and these could include credit to be allocated for resolving engineering risks associated with assurance. These metrics can also relate to compliance with standards associated with interlinked systems, as noted in the first scenario.
- *Architecture and component identification.* As noted earlier, the architecture definition is coupled with hazard identification and scope definition. The exceedingly high engineering risk for assurance and architecture in the after-the-fact scenario (assuming innovative architectural elements are required) is replaced with an up-front process of architecture modeling, supported by various early-validation techniques such as simulation, prototyping, and direct analysis (such as with model checking). Certain detail-level architectural commitments can be made incrementally. Progress metrics related to assurance-related engineering risk are refined and elaborated.
- *Component-level error and failure modeling.* A key difference is that the component-level modeling, combined with the supply-chain appraisal, provides an early feedback mechanism regarding engineering risks in the evolving architecture design. Risks can be assessed related not only to quality attributes and technical feasibility, but also to sourcing costs and risks. For example, choices might be made regarding opaque commercial components from a trusted source, custom components, wrapped untrusted components, and open-source components that afford stakeholders both visibility and the possibility of useful intervention (e.g., adding test cases, adapting APIs, adding features, etc.). This process can also lead to the early creation of unit test cases, analysis and instrumentation strategies, and other quality-related interventions in the component engineering process. Process metrics defined in earlier stages can inform allocation of resources in this stage of the process. The metrics are also refined as part of the incremental development process.
- *Supply-chain and development history appraisal.* See above. The committee notes that it is sometimes asserted that offshore development is intrinsically too dangerous. However, one could argue that badly managed onshore development by cleared individuals may be more dangerous than offshore development with best practices and evidence creation along with coding. A well-managed offshore approach may be feasible for many kinds of components when elements of the evolving best practice are adopted, such as (1) highly modular architectures enabling simplicity in interface specifications and concurrent development, (2) unit testing, regression testing, and code analysis, with results (and tests) delivered as evidence along with the code, (3) frequent builds, (4) best-practice configuration control, and (5) agile-style gating and process management.[28] Metrics can relate to a combination of adoption of best practices and production of separately verifiable evidence to support any assurance claims. As noted above, full line-by-line historical tracking of changes to a code base is now commonplace for development projects of all sizes. A key benefit of such tracking is that it provides full traceability not only among artifacts, but also to individual developers, which is useful for security and to assure that individual developers are fully up-to-date with best practices.

[28] Michael A. Cusumano, Alan MacCormack, Chris F. Kemerer, and William Crandall, 2009, *Critical Decisions in Software Development: Updating the State of the Practice, IEEE Software* 26(5):84-87. See also Alan MacCormack, Chris F. Kemerer, Michael Cusumano, and Bill Crandall, 2003, "Trade-offs Between Productivity and Quality in Selecting Software Development Practices," *IEEE Software* 20(5):78-85.

- *Analysis of architecture and component models.* This becomes part of the iterative early-stage process of refining architecture, quality attribute goals, functional scoping, and sourcing. If there are portions of the configuration that may create downstream challenges for evaluators, this is the opportunity to revisit design decisions to facilitate evaluation. For example, an engineer might suggest a change in programming language for a component in order to get a 5 percent speed up. At this stage of the process, that proposal can be considered in the light of how it might influence assurance with respect to quality attributes, interface compliance, correct functionality, and other factors. The decision could be made not to change the programming language, but rather to incentivize the vendor to make the next set of improvements in its compiler technology. These decisions are made using a multi-criteria metric approach, with criteria and weightings informed by the earlier stages.
- *Identify high-interest components.* Regardless of the front end of the process, there will be a set of high-interest components. Ideally, however, as a result of architecture decisions, the components in this category are not also opaque and untrusted. Regardless, components are prioritized on the basis of measured assurance-related engineering risk, with metrics as set forth in the earlier stages. This assessment will account for ongoing improvements in development technologies (e.g., languages, environments, traceability and knowledge management), assurance tools (e.g., test, inspection, analysis, and monitoring support), and modeling (for various quality attributes including usability).
- *Develop a component evaluation plan.* Allocate resources, set priorities, and identify assurance requirements on the basis of the analyses above. In this preventive scenario, this plan is largely a consequence of the early decisions regarding architecture, sourcing, hazards, and functional scope. Metrics are defined for resolution of engineering risk in all components (but particularly high-interest components), so progress can be assessed and credit assigned.
- *Assess individual components.* As above, this involves a combination of many different kinds of techniques. In the preventive scenario, component development can be done in a way that delivers not only code, but also a body of evidence including test cases, analysis results, in-place instrumentation and probes, and possibly also proofs of the most critical properties. (These proofs are analogous to what is now possible for type-safety and encapsulation integrity, which is now a ubiquitous analysis that is composable and scalable.) This supporting body of evidence that is delivered with code enables acceptance evaluators to verify claims very efficiently regarding quality attributes, functionality, or other properties critical to assurance. Metrics are developed to support co-production of component code and supporting evidence.
- *Select courses of action for custom components.* See above.
- *Select courses of action for opaque components and services.* For existing vendor components, the same considerations apply as in the previous scenario. If new code is to be developed in a proprietary environment, then there is the challenge of how to make an objective case (not based purely on trust) that the critical properties hold. Existing approaches rely on mutually trusted third parties (as in Common Criteria), but there may be other approaches whereby proof information is delivered in a semi-opaque fashion with the code.[29] Additionally, the proprietary developer could develop the code in a way that is designed to operate within a sandbox, in a separate process, or in another container—in this approach, the design is influenced by the need to tightly regulate control and data flows in and out of the contained component. Metrics would weight various criteria, with a long-term goal of diminishing the extent of reliance on trust vested in commercial vendors in favor of evidence production in support of explicit "assurability" claims.
- *Refine system-level assessment.* Given the high risks and costs of architectural change, in a preventive scenario, any adjustments to architecture are done incrementally as part of the overall process. Metrics would relate to the extent of architectural revisions necessary at each stage of the process.

[29] There is a wealth of literature on proof-carrying code and related techniques.

Conclusion

A key conclusion from these scenarios is the high importance of three factors: (1) The extremely high value of incorporating assurance considerations (including security considerations—see Box 4.1) into the full systems lifecycle starting with conceptualization, throughout development and acceptance evaluation, and into operations and evolution. (2) The strong influence of technology choices on the potential to succeed with assurance practices. (3) As a consequence, the value to DoD software producibility that comes from enhancements to critical technologies related to assurance, including both what is delivered (programming languages, infrastructure) and what is used during development (models and analytics, measurement and process support, tools and environments).

Recommendation 4-1: Effective incentives for preventive software assurance practices and production of evidence across the lifecycle should be instituted for prime contractors and throughout the supply chain.

This includes consideration of incentives regarding assurance for commercial vendor components, services, and infrastructure included in a system.

As illustrated in the scenario, when incentives are in place, there are emerging practices that can make significant differences in the outcomes, cost, and risk of assurance. The experience at Microsoft with the Lipner-Howard Security Development Lifecycle (SDL)[30] reinforces this—the lifecycle not only leads to better software but also incentivizes continuous improvement in assurance technologies and practices.

When ecosystems, vendor components, open-source components, and other commerical off-the-shelf (COTS) elements are employed, assurance practices usually necessitate the DoD to constantly revisit selection criteria and particular choices. The relative weighting among the various sourcing options, from an assurance standpoint, will differ from project to project, based on factors including transparency of the development process and of the product itself, either to the government or to third-parties. This affords opportunity to create incentives for commercial vendor components to include packaged assurance-related evidence somewhere between the two poles of "as is" and "fully Common Criteria certified." Advancement in research and practice could build on ideas already nascent in the research community regarding ways that the evidence could be packaged to support quality claims and to protect trade secrets or other proprietary technology embodied in the components.

Recommendation 4-2: The DoD should expand its research focus on and its investment in both fundamental and incremental advances in assurance-related software engineering technologies and practices.

This investment, if well managed, could have broad impact throughout the DoD supply chain. When both recommendations are implemented, a demand-pull is created for improved assurance practices and technologies.

Recommendation 4-3: The DoD should examine commercial best practices for more rapidly transitioning assurance-related best practices into development projects, including contracted custom development, supply-chain practice, and in-house development practice.

[30] Steve Lipner and Michael Howard, 2006, *The Security Development Lifecycle: A Process for Developing Demonstrably More Secure Software*, Redmond, WA: Microsoft Press.

Several leading vendors have developed explicit management models to accelerate the development of assurance-related technologies and practices, to validate them on selected projects, and to transition them rapidly into broader use.[31]

[31] Microsoft is well known for its aggressive use of development practices including process (the Security Development Lifecycle (SDL) noted earlier—see http://msdn.microsoft.com/en-us/library/ms995349.aspx) and analysis tools (such as SLAM, PreFast, and others—see, for example Thomas Ball, 2008, "The Verified Software Challenge: A Call for a Holistic Approach to Reliability," pp. 42-48 in *Verified Software: Theories, Tools, Experiments*, Bertrand Meye and Jim Woodcock, eds. Berlin: Springer-Verlag).

5

Reinvigorate DoD Software Engineering Research

In this chapter, the committee summarizes and recommends seven technology research areas as critical to the advancement of defense software productibility. These seven areas were identified by the committee on the basis of the following considerations:

- Priorities identified from the analysis reported in the foregoing chapters: architecture, incremental process, measurement, and assurance. This builds on extensive interviews with leaders from the DoD and industry and research regarding both challenges and potential opportunities for the DoD.
- Areas of potential technology and practice that might not otherwise develop sufficiently rapidly without direct investment from the DoD. Although other agencies are investing in areas related to software producibility, the focus and approach to investment do not sufficiently address the priorities as identified above.
- Potential for a fleshed-out program proposal to satisfy research management "feasibility" criteria such as the Heilmeier questions (see Box 5.1), which identify a set of "tests" for research program proposal[1]—that is, areas where investment most likely leads to a return that benefits the DoD.
- Areas not sufficiently addressed by other major federal research sponsors, including the Networking and Information Technology Research and Development (NITRD) agencies.

Prefacing this summary of areas recommended for future research investment is an exploration of the role of academic research in software producibility and a discussion of the impacts of past investments. The chapter also includes a brief discussion regarding effective practice for research program management to maximize impact while managing overall programmatic risk.[2]

[1] There are many versions of the questions; one such version can be found in Box 5.1.

[2] Indeed, there is a parallel between programmatic risk in the development of innovative software and programmatic risk in research program management. More important, perhaps, is the analogy between engineering risk in innovative software development and management risk in research program management. Several kinds of research management risk and various approaches to management risk mitigation are identified in Chapter 4 of National Research Council (NRC), 2002, *Information Technology Research, Innovation, and E-Government,* Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=10355. Last accessed August 20, 2010.

THE ROLE OF ACADEMIC RESEARCH IN SOFTWARE PRODUCIBILITY

The academic research community—along with a small number of industry research groups—has traditionally addressed many of the core technical problems related to software producibility. The academic value proposition has several direct components: The first is *workforce*. University graduates are the core of the engineering workforce. The most talented and highly trained graduates—those who contribute to innovation in a primary way—tend to come from PhD programs. More generally, the research community generates a steady supply of people—graduates at all levels—educated at the frontiers of current knowledge in important areas of specialization. The economics of these programs depend on externally funded research projects. That is, unlike bachelor's and master's enrollments, the production of PhD graduates by universities is in direct proportion to sponsored research. It is perhaps too obvious to point this out, but cleared individuals with top technical qualifications are most likely to be graduates of U.S. universities.

The second component is *new knowledge*. The style of computer science and software research, historically, has focused on the creation of scientific understanding that is both fundamental and applicable. This is in keeping with the "boundlessness" of software as described in Chapter 1.[3] Although industry plays a limited role in performing research relevant to fundamental open problems, there is no institution in the United States other than the research community, located primarily at universities, that focuses on broad and often non-appropriable advancements to knowledge that are directly relevant to practice. Indeed, major corporate labs that have historically supported non-appropriable and open-publication research as a significant part of their overall portfolios (such as Bell Labs and Xerox PARC) have been restructured or scaled back in recent years. This scaling back of private-sector research is due to numerous factors, including a loss by many players of safe monopoly status, analogous to that which enabled Bell Labs to thrive. This creates greater internal pressure on laboratory managers to create measurable return on investment (ROI) cases for research projects. This is particularly challenging for software producibility research, which is often focused on creating new measures of "return" rather than on incremental advances according to readily measurable criteria. This increases the significance of the role of academic research, government laboratories, and federally funded research and developments centers (FFRDCs). This is not to say that major research effort in software producibility is not underway in industry. At Microsoft and IBM, particularly, there is aggressive and forward-looking work in this area that is having significant influence across the industry.

Academic research and development (R&D) is also a major generator of entrepreneurial activity in information technology (IT).[4] The small companies in that sector have an important role in developing and market testing new ideas. The infrastructure to support these ventures is an important differentiator of the U.S. innovation system. This infrastructure includes university intellectual property and people supported by university R&D projects. These companies may sometimes disrupt the comfortable market structures of incumbent firms, but arguably not in the same way as do competition or foreign innovation. Regardless, weak incumbents tend to fall by the wayside when there is any disruption. Strong incumbents become stronger. This constant disruption is a characteristic of the more than half-century of IT innovation. It is essential that the DoD itself be effective as a strong incumbent that is capable of gaining strength through disruptive innovations, rather than being a victim (see below). The intelligence community's Disruptive Technology Office (DTO, now part of Intelligence Advanced Projects Research Agency[5]) can be presumed to have been founded upon this model.

A third area of value provided by university-based R&D (and industrial lab R&D as well) is *surprise reduction*. Computing technology is continuing to experience very rapid change, at a rate that has been

[3] This is the fundamental yet eventually useful knowledge in what Donald Stokes has called Pasteur's Quadrant. See Donald E. Stokes, 1997, *Pasteur's Quadrant—Basic Science and Technological Innovation*, Washington, DC: Brookings Institution Press.

[4] The committee uses "information technology" or "IT" to refer to the full range of computing and information technology areas in the scope of the NITRD multi-agency coordination activity (see http://www.nitrd.gov/ Last accessed August 20, 2010).

[5] See http://www.iarpa.gov/. Last accessed August 20, 2010.

BOX 5.1
Heilmeier Criteria

When George Heilmeier was DARPA director in the mid 1970s he developed a set of pithy questions to ask research program managers when they proposed new program ideas. That set of questions has persisted, and it continues to be applied in various forms by research managers everywhere. Here is a composite rendering of the questions, along with some commentary regarding research program management.

1. *What are you trying to do? Explain objectives using no jargon.* The scope of the project must be defined, as well as the key stakeholders in the outcome. The purpose of "no jargon," in part, is to assure that the scope and value can be described in ways that individuals outside the immediate field can appreciate the context and value of what is proposed.
2. *How is it done today? What are the limits of current practice?* This is an accounting of the baseline state, the value it delivers, the limits on what can be done in the present configuration, and, to some extent, the pain experience as a consequence of those limits.
3. *What's new in your approach? Why do you think it will be successful?* Often the novelty is less in the form of a dramatically "new idea," but rather in the convergence of existing ideas and new developments elsewhere in the field. A cynical view of "cloud computing," for example, is that it is a delivery on the dream of "utility computing" articulated in the early 1960s at the dawn of the era of timesharing. Cloud, of course, takes this idea many steps forward in scalability, capability, and other ways. In other words, it is less important that the idea be "novel," but rather timely, potentially game changing, and feasible. Feasibility, in this context, does not mean free of risk, but rather that the dependencies on infrastructure and other elements of the package are realistic. Feasibility also means that there are potential research performers who have the means and motive to engage on the topic. For academic research, this means the ability to build a capable team of PhD students, engineering staff as required, potential transition partners, collaborators at other institutions, etc.
4. *If you're successful, what difference will it make? To whom?* This is an identification of stakeholders, and in addition an indication of potential pathways from research results to impact. For many research projects related to computing and software, those pathways can be complex. These complexities are discussed in the

undiminished for several decades and perhaps is accelerating because of a now-global involvement in advancing IT. Given the rapid change intrinsic to IT, the research community (in academia and in industry, especially start-up companies) serves not only as a source of solutions to the hardest problems, a source of new concepts and ideas, and a source of trained people with high levels of expertise, but also as a bellwether, in the sense that it anticipates and provides early warning of important technological changes. For software, the potential for surprise is heightened by a combination of the rapid growth of globalization, the concurrent movement up the value chain of places to which R&D has been outsourced, and the explicit investments from national governments and the European Union in advancing national technological capability. Given the role of externalities in IT economics, it is not unreasonable to expect the innovation center of gravity to change rapidly in many key areas, which could shift control in critical areas of the technology ecosystems described above. This is already happening in several areas of IT infrastructure, such as chip manufacturing. In this sense, the research community has a critical role in defense-critical areas that are experiencing rapid change. A consequence of this role is the availability of top talent to address critical software-related defense problems as they arise.

The fourth component of the academic R&D value proposition is *non-appropriable invention*, as described in Chapter 1. This is one of the several forms of innovation carried out by the university

NRC "tire tracks" reports.[1] For software, the path often connects the research results to the DoD through the development of commercial capabilities, where private investment takes a promising research idea and matures it to the point that it can be adopted by development teams. This adoption could be by software development teams in defense contractors or it could be by development teams creating commercial products or services. For example, the reliability of DoD desktop computers undeniably was improved, quite dramatically, as a result of the improvements made by Microsoft to the process of development and evaluation for device driver code enabled by the SLAM tool (described elsewhere in this chapter), which in turn were enabled by research sponsorship from DARPA and NSF. In addition to defining the impact, there is value in understanding not only those stakeholders who will benefit, but also those who may be disrupted in other ways.

5. *What are the risks and the payoffs?* This is not only an accounting of the familiar "risk/reward" model, but also an indication of what are the principal uncertainties, how (and when) they might be mitigated, and what are the rewards for success in resolving those uncertainties.[2]

6. *How much will it cost? How long will it take?* An important question is whether there are specific cost thresholds. For certain physics experiments, for example, either the apparatus can be built, or not. But for other kinds of research there may be more of a "gentle slope" of payoff as a function of level of effort. The answer to the questions of cost and schedule, therefore, should not only be specific numbers, but also, in many cases, should provide a description of a function that maps resources to results.

7. *What are the midterm and final "exams" to assess progress?* It is essential that there be ways to assess progress, not only at the end of a project, but also at milestones along the way. (This is analogous to the idea of "early validation" of requirements, architecture, design, etc., as a way to reduce engineering risk in software.) In many research areas, quantitative measures of progress are lacking or, indeed, their formulation is itself the subject of research. For this reason, in some challenging research areas the identification

[1] See National Research Council (NRC), 1995, *Evolving the High Performance Computing and Communications Initiative*, Washington, DC: National Academy Press; and NRC, 2003, *Innovation in Information Technology*, Washington, DC: National Academies Press.

[2] An inventory of "engineering" risks related to research program management is in the NRC report on E-Government National Research Council, 2002, *Information Technology Research, Innovation, and E-Government*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=10355. Last accessed August 3, 2010.

research community. In a market economy, with internal ROI cases prerequisite for R&D investment inside firms, this is a role most appropriate to universities and similar institutions—of course firms often carry out or sponsor such innovation for a variety of reasons, but it is not their core purpose. For IT in particular, such R&D is essential to national competitiveness and to increases in market-wide value. Although the openness of university research is sometimes considered a negative factor with respect to the advancement of technology for national security, it is also the case that universities have unique incentives, unlike industry, to advance the discipline even when the hard-won results are non-appropriable or difficult to fully appropriate. As noted above, it is evident from the history of the field that the advancement of IT and software producibility disproportionately depends on this kind of technology advance. Of course, universities also create an enormous body of appropriable intellectual property that has the potential to be transitioned into practice.

Finding 5-1: Academic research and development continues to be the principal means for developing the most highly skilled members of the software workforce, including those who will train the next generation of leaders, and for stimulating the entrepreneurial activity that leads to disruptive innovation in the information technology industry. Both academic and industry labs are creating

the fundamental advances in knowledge that are needed to drive innovation leadership in new technologies and to advance software technologies that are broadly applicable across industry and the DoD supply chain.

DoD Influence on Academic R&D

The overall directions and priorities for sponsored research that leads to university-originated invention are greatly influenced by funding levels and agency priorities. For example, the Defense Advanced Research Project Agency's (DARPA's) deliberately strong relationship with the IT research community, which began in the 1960s and endured for nearly 40 years, has had profound influence on IT research priorities, the overall culture of computer science research, and the massive economic and national outcomes. This is documented in multiple NRC reports relating to the innovation pipeline for IT, which trace the origins of a broad set of specific IT innovations, each of which has led to a multi-billion dollar market.[6]

Data available from NITRD and other sources indicate that there has been a significant reduction in federally sponsored research related to software producibility as well as to high-confidence software and systems (see Box 1.5). Furthermore, it is the committee's impression that in recent years many of the researchers in these areas have moved into other fields or scaled down their research efforts as a result of, among other things, the DoD's having shifted funding away from software-related R&D, apparently on the assumption that industry can address the problems without government intervention. As stated previously, industry generally has less incentive to produce the fundamental advances in knowledge that enable disruptive advances in practice, building on fundamental advances but less often creating them. The impact of R&D cutbacks generally (excluding health-related R&D) has been noted by the top officers of major IT firms that depend on a flow of innovation and talent.

Academic R&D, Looking Forward

There are some challenges to proceeding with a new program for academic R&D related to software-intensive systems producibility. These challenges relate generally to saliency, realism, and risk. University researchers and faculty tend to be aware of broadly needed advances, but they do not always have adequate visibility into the full range of issues created by leading demands for large-scale, complex industrial and military systems. This awareness is hindered by many things, including national security classification, restricted research constraints, professional connectivity, and cost, in the sense of time and effort required to move up the learning curve. In a different domain, DARPA took a positive step in this regard by initiating the DARPA Computer Science Study Group, wherein junior faculty are given clearances and so are able to gain direct exposure to military challenge problems. Several specific DoD programs have undertaken similar efforts to give faculty a domain exposure, often with great success. One example from the 1990s is the Command and Control University (C2U) created by the Command Post of the Future (CPOF) program, which not only gave researchers access to military challenges, but also led to collaborations yielding new innovation in system concepts.[7]

With respect to ensuring that researchers have access to problems at scale, companies such as Google and Yahoo!, and national laboratories such as Los Alamos, have developed collaborative programs to expose faculty and graduate students to high-performance computing systems, large datasets, and the software approaches being taken with those systems. These companies, like the DoD, have worked out

[6] See NRC, 2003, *Innovation in Information Technology*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=10795. Last accessed August 20, 2010. Also see the predecessor report NRC, 1995, *Evolving the High Performance Computing and Communications Initiative*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=4948. Last accessed August 20, 2010.

[7] The committee understands that prototype systems from this program are now deployed in Iraq.

a level of exposure that enables researchers to engage productively without compromising core intellectual property. The DoD has a track record of success in this regard as well.

For software producibility research, a different kind of access is needed. Certainly, the success of large-scale production-quality open source has afforded researchers great opportunity not only to experiment with large code bases, but also to undertake longitudinal and organizational analyses of larger projects. This has been enabled by the sophistication of the tools—code management systems, defect databases, designs and models, test cases. These projects are comparable in scale and functionality to commercial software and have greatly assisted the software engineering community in its research. Additionally, commercial firms are affording researchers greater access to proprietary code bases for experimentation and analysis. An early and significant example is work by Gail Murphy in which she assessed consistency of an as-built code base with architectural intent.[8] She studied both an open source project and a proprietary project. If security and commercial ownership issues could be resolved (perhaps by clearing selected researchers), members of the research community would benefit greatly from access to DoD-related artifacts, including surrogates and "sanitized" artifacts that omit critical algorithms and/or data. Regardless of access, the committee recommends improved data collection to support analysis (see Recommendation 2-2).

INVESTING IN RESEARCH IN SOFTWARE PRODUCIBILITY

The Impact of Past Investments

Software development has changed and, for the most part, improved considerably during the past several decades. Software systems have grown in size and complexity and are now an integrated component of every aspect of our society, including finance, transportation, communication, and health care. Since the 1960s, Moore's Law has correctly predicted the tremendous growth in the number of transistors on chips and, generally speaking, the extent of hardware-delivered computing power. An analogous growth has occurred in the size and power of software systems if machine-level instructions, rather than transistors, are the measure of growth.[9,10,11] Today's systems are built using high-level languages and numerous software library components, developed using sophisticated tools and frameworks, and executed with powerful runtime support capabilities.

Research in software engineering, programming technologies, and other areas of computer science has been a catalyst for many of these advances. Nearly all of this research was undertaken at research universities as part of federal programs led by DARPA, the National Science Foundation (NSF), and the Service basic (category 6.1) research programs of the Office of Naval Research, Air Force Office of Science Research, and Army Research Office.

Three illustrations of the impact of federal sponsorship (in academia and industry) that is specifically related to software engineering are presented in Box 5.2. These illustrations, drawn from a study undertaken by Osterweil et al.,[12] complement the analyses of the NRC reports cited above relating to research impacts on practice and on the IT economy.

[8] Gail Murphy, 1995, "Software Reflexion Models: Bridging the Gap Between Source and High-level Models," *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, Washington, DC, October 10-13, pp. 18-28.

[9] Barry Boehm, 1999, "Managing Software Productivity and Reuse," *IEEE Computer* September, 32(9):111-113.

[10] Mary Shaw, 2002, "The Tyranny of Transistors: What Counts about Software?" *Proceedings of the Fourth Workshop on Economics-Driven Software Engineering Research*, IEEE Computer Society, Orlando, FL, May 19-25, pp. 49-51.

[11] Barry Boehm, 2006, "A View of 20th and 21st Century Software engineering," *Proceedings of the 28th International Conference on Software Engineering*, ACM, Shanghai, China, May 20-28, pp. 12-29.

[12] Leon J. Osterweil, Carlo Ghezzi, Jeff Kramer, Alexander L. Wolf, 2008, "Determining the Impact of Software Engineering Research on Practice," *IEEE Computer* 41(3):39-49.

Challenges and Opportunities for Investment

Notwithstanding the enormous payoffs from past investments in software research, making the case for future research investments in software producibility faces a number of challenges, rooted largely in the nature of software development as a field of study. All scientific research fields face challenges in justifying future investments, but the unique characteristics of software and the dynamics of knowledge creation in software producibility create particular challenges for this field. There are, however, opportunities based on developments in the technology, in the overall environment of practice, and in the improvement of scientific practice. These challenges and opportunities influence the application of the criteria summarized at the outset of this chapter. Below are a few examples of influences, both positive and negative:

- *Maturation of the discipline.* Many researchers will agree that, as a discipline, software engineering research has matured considerably in the past decade. This is a consequence of both improved research methods and improved circumstances. The circumstances include a vast improvement in access to large bodies of code, both through large-scale open-source projects and through improvement in researcher access to proprietary code bases. An additional circumstance is the emergence of highly capable tools, including source-code management systems, development environments, analysis frameworks, etc., that afford researchers opportunity to conduct experiments at meaningful levels of scale. The effect is that it is more often possible for software engineering researchers to give satisfactory responses to the Heilmeier questions (see Box 5.1). At the same time, software engineering practice remains behind the state of the art in research. As discussed in Chapter 1, software development remains more akin to a craft than to an engineering discipline, in which the productivity and trustworthiness of system development rest on fundamental and well-validated principles, practices, and technologies. And it is still the case that even sanitized representative software artifacts are not available for academic analysis in many defense areas.
- *Diffusion pathways and timescale.* Many of the results of software research are broadly applicable and provide for enabling technologies and methods useful in a range of specific application domains. Breadth of applicability is valued in research, but it is also double-edged from the standpoint of sponsors. First, there is a greater chance that results may diffuse to adversaries as well as to collaborators. Second, there is a commons problem: because the benefits are broad, no particular stakeholder can justify the investments needed to produce them. Thus, for example, DoD Service R&D programs tend to focus much more on Service-specific technologies than on common-benefit software technology. Twenty years ago, the Service Laboratories played a significant part in maturing and transitioning software producibility technology, but the "tragedy of the commons" has virtually dried up this key channel. Moreover, advances in software producibility *very often are enabling advances* rather than *being advances* of immediate use in particular products. Better techniques for identifying, diagnosing, and repairing software faults, for example, enable production of better systems but are not directly used in particular software products. The value of such advances is thus often hard to quantify precisely for any single advance, or from the perspective of any single program. Yet when integrated over longer periods of time and in terms of impacts on many engineering products, the benefits of the stream of advances emerging from software research are very clear (as summarized above). In the case of defense software producibility, there are clear drivers of defense software "leading demand," and there are ways that the DoD can invest in and realize benefits earlier and more effectively than can potential adversaries. Moreover the DoD remains a major beneficiary of the longer-term production of software producibility knowledge.
- *Novelty of ideas.* It is noted earlier in this chapter that cloud computing, taken broadly, is really a manifestation of a half-century-old idea of "utility computing" that has just now become feasible due to the positions of the various exponential curves that model processor, storage, and communications capabilities and costs—as well as enabling engineering, management, and business innovation. This account is a bit simplistic, obviously, but it makes an essential point: The specific novelty of an idea

BOX 5.2
Three Examples of the Impact of Past Investments

1. *From bug detection to lightweight formal methods.* Bugs have plagued software since before there were computers,[1] and researchers have been actively working on developing tools to help detect and prevent errors in software systems for at least half a century. Early compilers focused on syntactic errors and simple debugging support, but soon tools were developed to detect more complex semantic errors. Simple definition-reference bug detection techniques[2,3] were followed by more sophisticated approaches.[4,5,6] Programming languages such as Ada, Java, and C# incorporated some of these concepts directly into the language, and thus, for example, checked for array indexes being out of bounds during compilation and added runtime checking only when necessary. This work laid the foundation for a range of model checking and program analysis tools that are now emerging at companies like Microsoft and Google as these companies increase their concern for secure, high-quality systems. Systems such as Microsoft's SLAM and PreFAST are based upon the research advances funded by the federal government. For example, a report on SLAM states, "The project used and extended ideas from symbolic model checking, program analysis and theorem proving."[7] Those ideas emerged from academic research performed years earlier related to model checking and binary decision diagrams, and indeed Edmund Clarke won the 2008 Turing Award for his work on model checking. The authors of this tool, which has been is credited with significantly reducing the incidence of "blue screen" system crashes, were awarded the 2009 Microsoft Engineering Excellence, a success that represents the culmination of federally funded research from the 1970s through the 1990s.

Early research on software testing advocated for coverage measures, such as statement and branch coverage, and tools were developed for symbolically executing paths in programs and automatically generating test cases to satisfy such measures[8,9,10] The storage and speed of the machines at that time made this approach impractical, but advances in hardware combined with continued research advances in lightweight reasoning engines and higher-level languages have now made coverage monitoring a

---

[1] Letters from Ada Lovelace to Charles Babbage discussing programming errors are mentioned in Grady Booch and Doug Bryan, 1993, Software Engineering with ADA, 3rd Ed., Boston: Addison-Wesley Professional. Also see Grace Murray Hopper's note in the log for the Aiken Mark II in 1947 in Grace Murray Hopper, 1981, "The First Bug," Annals of the History of Computing 3(3):285-286, 1981.

[2] Leon J. Osterweil and Lloyd D. Fosdick, 1976, "Some Experience with DAVE: A Fortran Program Analyzer," in *Proceedings of the National Computer Conference and Exposition*, ACM, New York, NY, June 7-10, pp. 909-915.

[3] Barabara G. Ryder, 1974, "The pfort Verifier," *Software: Practice and Experience* 4(4):359-377.

[4] Kurt M. Olender and Leon J. Osterweil, 1990,"Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation," *IEEE Transactions on Software Engineering* 16(3):268-280.

[5] Edmund M. Clarke and E. Allen Emerson, 1981, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic." Pp. 52-71 in *Logic of Programs: Workshop Lecture Notes in Computer Science* 131, Berlin: Springer.

[6] Gerard J. Holzmann, 1997, "The Model Checker SPIN," *IEEE Transactions on Software Engineering* 23(5): 279-295.

[7] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani, 2004, "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft," *Lecture Notes in Computer Science (LNCS)* 2999:1-20; Eerke A. Boiten, John, Derrick, and Graeme, Smith, eds., 2004, *Fourth International Conference on Integrated Formal Methods (IFM 2004)*, Canterbury, Kent, England, April 4-7. Researchers at Microsoft have stated that the majority of the "blue screen of death" errors evident in the 1990s were attributed to problems that could have been prevented with this analysis tool.

[8] Lori A. Clarke, 1976, "A Program Testing System," in *Proceedings of the 1976 ACM Annual Conference,* ACM, Houston, TX, October 20-22, pp. 488-491.

[9] James C. King, 1975, "A New Approach to Program Testing," in *Proceedings of the International Conference on Reliable Software*, ACM, Los Angeles, CA, April 21-23, pp. 228-233.

[10] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt, 1975, "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution," in *Proceedings of the International Conference on Reliable Software*, ACM, Los Angeles, CA, April 21-23, pp. 234-245.

continued

BOX 5.2 continued

common industrial practice, along with sophisticated support for test case generation.[11] Similarly, current trends in testing, such as the "Test First" approach widely adopted by agile software development teams (and tools such as the JUnit unit-testing framework) owe their foundation to early work on test case descriptions and automated execution,[12,13] again based on U.S. government-funded research.

Programming language development has also been strongly influenced by work in analysis of software systems, as noted above with Ada and its support for type safety and automated bounds checking. Although Ada was not a broad commercial success for various political, programmatic, and social-economic reasons, it is recognized as the direct ancestor of Java, which is widely adopted partly because of its embodiment of lessons from type theory, program analysis, and programming environment research. These lessons enabled Java to support richly capable libraries and software frameworks (as noted in Chapter 1). The C# language from Microsoft builds on similar foundations, and all three languages provide a stronger foundation for writing secure and high-quality code.

2. *From development environments to software architectures to domain-specific frameworks*. The success of Java is also partly due to the recognition of the importance of an interactive development environment (IDE). Early development environments were language centric, such as Interlisp[14] from 1981, but continued government-supported research, such as Field[15] and Arcadia,[16] advocated for looser interaction models and broad support for interoperability. This led to work on common data interchange models,[17]the forerunners to XML and all its variants, multi-language virtual machine models such as the Java Virtual Machine, and common interoperability protocols, such as Java's remote method invocation (RMI) and certain features of Microsoft's .NET framework. These advances, combined with the language principles, enabled the development of modern integrated development environments (IDEs) such as Microsoft's Visual Studio and Eclipse, originally developed by IBM but later released to open source.

As software systems continued to grow in size and complexity, software engineering research broadened from algorithm and data structure design to include software architecture issues[18,19] and the recog-

---

[11] Dorota Huizinga and Adam Kolawa, 2007, *Automated Defect Prevention: Best Practices in Software Management,* Hoboken, NJ: Wiley-IEEE Computer Society Press.

[12] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini, 1998, "Evaluating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering* 24(8):586-601.

[13] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini, 1997, "Choosing a Testing Method to Deliver Reliability," in *Proceedings of the 19th International Conference on Software Engineering,* ACM, Boston, MA, May 17-23, 1997, pp. 68-78.

[14] Warren Teitelman and Larry Masinter, 1981,"The Interlisp Programming Environment," *Computer* 14(4):25-33.

[15] Steven P. Reiss, 1990, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software* 7(4):57-66.

[16] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young, 1989, "Foundations for the Arcadia Environment Architecture," *ACM SIGSOFT Software Engineering Notes* 24(2):1-13.

[17] David Alex Lamb, 1987, "IDL: Sharing Intermediate Representations," *ACM Transactions on Programming Languages and Systems* 9(3):297-318.

[18] Mary Shaw and David Garlan, 1996, *Software Architecture Perspectives on an Emerging Discipline*, Upper Saddle River, NJ: Prentice Hall.

[19] Dewayne E. Perry and Alexander L. Wolf, 1992, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes* 17(4):40-52.

nition of common styles and design patterns.[20] Work in software architecture was also an enabler of the development of high-level frameworks, such as Service Oriented Architectures[21] electronic enterprise systems, the backbone of current e-business. Commercial architecture standards such as REST[22] derive from government-supported software architecture research.

3. *From the waterfall to an agile compromise.* Early software developers often viewed themselves as independent artisans because they worked individually or in very small groups. The reality of the complexity of the systems that were being developed, the long-term duration, the vast resources required, and the large percentage of unsuccessful projects, led to the realization that large software system development needed to be supported by a carefully managed process. Early process models, and particularly the waterfall model,[23] were developed as organizing frameworks to help organize the considerable pre-implementation and modeling work, and within them were identified the major software development phases. The actual flow from phase to phase was sometimes interpreted overly simplistically, leading to process models (e.g., the DoD 2167A standard) that are now considered cumbersome and overly rigid. Software leaders in academia and industry, such as Belady, Lehman, Mills, Boehm, and others, argued for more reasoned development models that incorporated risk assessment and incremental, evolutionary development.[24,25,26] These models contained the seeds of the iterative ideas that now are nearly ubiquitously adopted by small development teams throughout industry. These were documented in the case of Microsoft by Cusumano and Selby[27] and in the now extensive literature of small-team iterative methods under rubrics such as extreme, agile, scrum, TSP, and others. These methods are quite aggressively driving the development of tools to better support team activity including coordination across teams to support larger projects. Concepts including code refactoring, short development sprints, and continuous integration are now accepted practices. However, most agile practices have serious assurance and scalability problems,[28] and need to be used selectively in large mission-critical systems or systems of systems.

---

[20] Martin Fowler, 2002, *Patterns of Enterprise Application Architecture*, Boston: Addison-Wesley Longman Publishing.

[21] Michael Bell, 2008, "Introduction to Service-Oriented Modeling," *Service-Oriented Modeling: Service Analysis, Design, and Architecture,* Hoboken, NJ: Wiley & Sons.

[22] Roy T. Fielding and Richard N. Taylor, 2002, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology* 2(2):115-150.

[23] Winston W. Royce, 1970, "Managing the Development of Large Software Systems: Concepts and Techniques," *Technical Papers of Western Electronic Show and Convention (WesCon),* August 25-28, Los Angeles, CA.

[24] Laszlo Belady and Meir M. Lehman, 1985, *Program Evolution Processes of Software Change*, London, UK: Academic Press.

[25] Barry Boehm, 1986, "A Spiral Model of Software Development and Enhancement," *ACM SIGSOFT Software Engineering Notes* 11(4):14-24.

[26] Harlan Mills, 1991, "Cleanroom Engineering," *American Programmer,* May, pp. 31-37.

[27] Michael A. Cusumano and Richard W. Selby, 1995, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People,* New York: Harper Collins Business.

[28] Barry Boehm and Richard Turner, 2004, *Balancing Agility and Discipline*, Boston: Addison-Wesley.

may matter much less than the timeliness of the idea and the readiness of the environment to address it in a successful way. Many old ideas, considered as failing concepts, resurfaced years later at the "right time" and made a significant difference. In other words, the key question is not so much, What are the new ideas? but rather, What are the ideas whose time has come?

- *Measurement of effectiveness and performance.* The challenges of software measurement as discussed in the previous chapters—with respect to process measures, architecture evaluation, evidence to support assurance, and overall extent of system capability—apply also to software engineering research. We lack, for example, good ways to measure the impact of any specific research result on software quality, which stems in part from the lack of good measures of software quality. Without reliable, validated measures it is hard to quantify the impact of innovations in software producibility, even those that are widely credited with improving quality, such as the introduction of strong typing into programming languages or traceability in software-development databases. This is analogous to the productivity paradox, recently resolved.[13] Because software is an enabling technology—a building material rather than a built structure—it may not fit with research program management models that focus on production of artifacts with immediately, clearly, and decisively measurable value.
- *Timescale for impact.* Frequently, it is only after a significant research investment has been made and proof of concept demonstrated that industry has stepped in to transition a new concept into a commercial or in-house product. Also, there are many novel products/services that result from multiple, independent research results, none of which is decisive in isolation, but which when creatively combined lead to breakthroughs. Although it may appear that a new development emerged overnight, further inspection usually reveals decades of breakthroughs and incremental advances and insights, primarily funded from federal grants, before a new approach becomes commonly accepted and widely available. CSTB's 2003 report *Innovation in Information Technology* reinforces this point. It states, "One of the most important messages … is the long, unpredictable incubation period—requiring steady work and funding—between initial exploration and commercial deployment. Starting a project that requires considerable time often seems risky, but the payoff from successes justifies backing researchers who have vision."

AREAS FOR FUTURE RESEARCH INVESTMENT

In this section, the committee identifies seven areas for potential future research investment and, for each area, a set of specific topics that the committee identifies as both promising and especially relevant to defense software producibility. These selections are made on the basis of the criteria outlined at the beginning of this chapter. The descriptions summarize scope, challenges, ideas, and pathways to impact. But, obviously, these descriptions are not (even summary) program plans—the development of program plans from technical descriptions requires consideration of the various program management risk issues,[14] development of management processes and plans on the basis of the risk identification, identification of collaborating stakeholders, and other program management functions. In the development of program plans, choices must be made regarding scale of the research endeavor and the extent of prototype engineering, field validation, and other activities that are required to assess the value of emerging research results. In some areas, a larger number of smaller projects may be most effective, while in other areas more experimental engineering is required and the research goals may be best addressed

[13]This is analogous to the so-called "productivity paradox," according to which economists struggled to account for the productivity benefits that accrued from investments made by firms in IT. The productivity improvements due to IT are now identified, but for a long time there was speculation regarding whether the issue was productivity or the ability to measure particular influences on productivity. (This issue is also taken up in Chapter 1.)

[14]An inventory of risk issues for research program management appears in Chapter 4 of NRC, 2002, *Information Technology Research, Innovation, and E-Government*, Washington, DC: National Academy Press. Available online at http://www.nap.edu/catalog.php?record_id=10355. Last accessed August 20, 2010.

through a small set of larger and more integrated projects.[15] Also in the development of program plans, choices must be made regarding the degree to which an agency program focuses on a particular solution strategy—rather than posing a problem and soliciting a diversity of potential solution approaches, many of which may not have been anticipated when the problem was posed and the program formulated. Particularly in software research, where the development of new metaphors and models is essential to progress, this latter approach can be very valuable.[16]

The descriptions, rather, serve primarily as a summary of points made in earlier chapters relating to technology advances that would "make a difference" in software producibility (and that meet the criteria). The committee offers them as recommended focal points for renewed investment in defense software producibility.

Area 1. Architecture Modeling and Architectural Analysis

As noted throughout this report, improvements in the ability of the DoD to manage system design, evaluation, development, and evolution at the architectural level are a key to improved software producibility. For precedented systems, such advances would mean having and using documented, validated architectures and making good ecosystems choices. Improvements here can increase the value and flexibility of libraries and frameworks, and can facilitate their use through modeling and validation, for example. For innovative systems (this report's principal focus), good architecture choices are often the keys to successful development and are significant to the scaling up and interlinking of systems, process management, enabling incremental practices, assurance, and reduction of diverse kinds of engineering risks related to design, interoperation, and supply-chain choices. Because the DoD benefits greatly from interlinked systems (net-centric, ultra-scale, systems of systems), advances in architecture-related capabilities make a greater difference both in potential to achieve systems capability and in ability to effectively manage architecture-related engineering risks. Yet, despite major advances in knowledge of software and system architecture, the state of knowledge and certainly the state of technology and practice today are inadequate to support DoD needs in this area, even for precedented systems. DoD success in software-intensive systems producibility depends on future research results in this area, and the transitioning of such results into useful notations, technologies, practices, and rules. The committee identifies three principal goals for architecture research.

Goal 1.1: Facilitate Mission-Oriented Modular Architectures

A good example of mission-oriented modular architecture is the decoupling of sensors, battle command, and weapons release. These functions, co-located in a tank, battleship, or fighter aircraft for example, not only can be separated geographically, but also can be shared across multiple battlefield functions. This has a near-irresistible value, analogous to Metcalfe's Law for network-structured systems.[17] This is part of the compelling rationale for goals associated with the Army Future Combat Systems (FCS) and Theater Ballistic Missile Defense (BMD) models with net-centric approaches, intelligence linking, and the like. In this model, a shooter can be guided by a multitude of geographically dispersed sensors, and unmanned sensors and shooters can be positioned at dangerous locations

[15] DARPA, for example, has used both approaches to advantage over the years.

[16] This is "solution risk" as described in Chapter 4 of NRC, 2002, *Information Technology Research, Innovation, and E-Government*, Washington, DC: National Academy Press. Available online at http://www.nap.edu/catalog.php?record_id=10355. Last accessed August 20, 2010.

[17] Metcalfe's Law asserts that the aggregate value of a network to its members grows with the square of the number n of members—proportional to the number of edges in a complete graph of size n. This is a folkloric explanation of why the pressure to combine networks (as in the original *inter*net, but also for instant message interoperation, convergence of fax standards, etc.) is so difficult for operators to resist, even when it creates business risks through loss of lock-in.

through the use of autonomous and teleoperated vehicles. The model thus affords tremendous power and agility to theater commanders.

With respect to this research issue: Because the architecture is fundamentally driven by interoperability and integration requirements, effective management of architecture can be a great enabler for joint (multiple military services, including air, land, sea, space, and cyber) and combined (international and coalition) warfare. But from the standpoint of systems engineering, the power comes at a significant price, which is the high level of complexity and engineering risk that comes from the extent of coupling and operational flexibility required among the multitude of sensors, weapons, and battle command centers. For example, how can architectures be developed and validated to support the kind of local autonomy necessary for a vehicle to navigate effectively over mixed terrain? How can "unanticipated requirements be anticipated" such as command and control for rapidly assembled coalitions, for example, to address a natural disaster? How can software and systems architectures be evolved, for example, as algorithms and machine-learning capabilities improve? Moreover, by specifying interfaces where testing or measurement is possible, by defining reusable components, and by separating critical from noncritical parts of the system, architecture plays an essential role in assurance. What happens when a vehicle or platform is compromised? How is resiliency built into the architecture to avoid a deliberately stimulated cascading failure?

Architecture is more than a "top down" laying out of systems structure or theoretical contemplation of design possibilities. The skills of a software architect in trading off diverse considerations to fix on essential design commitments is described by the Roman architect Vitruvius (ca. 15 BC):

> The architect should be equipped with knowledge of many branches of study and varied kinds of learning, for it is by his judgment that all work done by the other arts is put to the test. This service of his is the child of theory and practice. Practice is the continuous and regular exercise of employment where manual work is done with any necessary material according to the design of the drawing. Theory, on the other hand is the ability to demonstrate and explain things wrought in accordance with technical skills and method. It follows, therefore, that architects who have aimed at acquiring manual skills without theory have not been able to reach a position of authority to correspond with their pain, while those who relied only on theories and learning were obviously hunting the shadow, not the substance. But those who have mastered both, like men equipped in full armor, soon acquire influence and attain their purpose.

There are several specific challenges associated with this goal:

- *Architectural decisions.* Architectural decision making is driven by the combined consideration of multiple interacting factors. Some factors derive from stakeholder needs—these are functional scope and quality attributes such as degree of assurance needed, operational safety and security, design evolvability, online adaptability, performance, cost, etc. Other factors are internal factors, reflecting the interdependency of the various dimensions of architectural decision making. For today's major applications, for example, a diversity of architectural styles is induced by sets of interrelated decisions concerning the combination of frameworks, platforms, and middleware to be adopted. Advancing architecture into a more scientific activity requires improvement in our understanding of architectures as sets of critical and dynamic (and internal and external) parameter values subject to complex constraints and dependences.
- *Architecture scalability and evolvability.* Current architecture capabilities do not scale up to representing and evolving architecture models across multiple systems, multiple subcontractor levels, and multiple increments. They do not do well at such needed functions as change impact analysis or multi-version change propagation for large-scale systems or systems of systems.
- *Architectural measures.* In return for investments in architecture, one expects to gain predictable, quantifiable advantages in both system development and operation. This is particularly significant, as cost statistics show that architecture decisions account for greater proportions of overall cost as systems scale up in size and complexity. As architecture scales up, modularity becomes an increasingly crucial issue, for example. Decisions in this area are among the most consequential yet least well understood in

any major project. There are well-understood consequences for system trustworthiness (e.g., through the isolation of critical elements), producibility, flexibility, and adaptability. Less modularity makes assurance more elusive, and it makes changes more costly and risky. The research challenge is to develop new techniques for architectural modeling and analysis that focus on various measures of modularity and interlinking among system elements.

- *Architectural modeling and evaluation.* How can architectural models be expressed to support such diverse architecture-level analyses prior to the full development of code? What kinds of analytics can be developed, including simulation, static analyses of various kinds, model checking, and other analyses. What kind of traceability support can be created to connect architectural representations to representations of requirements and other stakeholder concerns, on one hand, and to the more detailed concerns of system design, construction, and governance of development and change, on the other?
- *Architecture compliance.* How can tools be developed (for increasingly complex architecture models and styles) to assist designers, developers, and requirements engineers in assessing, on an ongoing basis, the consistency of their models with architectural models? This is complicated by issues related to framework design, concurrency, and other issues. For example, a framework or application programming interface (API) may expect to receive an object not only of a particular type (e.g., "file handle") but also in a particular state ("open"). This is not well addressed in current programming languages or architectural models.

Goal 1.2: Facilitate Architecture-Aware Systems Management

Management of architecture aligns with management of sourcing of components and infrastructure, with system development and evolution, and also with definition of mission processes (or business processes). Such alignment, or *congruence* (which refers specifically to the relationship of architecture structure with organization structure),[18] is essential to managing the coordinated scaling up and evolution of systems, organizations, and the mission processes supported. It is the IT-business convergence that is a consideration for many corporate chief information officer (CIO) organizations and that is also a key to success for many IT-enabled firms.[19]

Challenges associated with this goal are as follows:

- *Models of congruence.* As architecture models are enriched, models for modeling and managing congruence become more complex and technically involved.
- *Enriched software supply chains.* Supply-chain structure is only increasing in richness and complexity, and it is further complicated by the greater extent of intertwining of iterative processes across producer/consumer boundaries. What architecture-level interventions could facilitate assessment, across a supply chain, of consistency of an evolving system with its defined architectural intent?
- *Ecosystems and infrastructure.* The DoD is unavoidably a participant in diverse commercial ecosystems. What architectural practices can assist in lessening the engineering risks associated with this involvement? For example, how can notions of technical software and system architecture be extended, adapted, or improved to enable better design and performance of the socio-technical ecosystems that surround, develop, and use complex systems?
- *Incompatible hardware and software architectural relationships.* As discussed in Chapter 2, many systems architectures are organized into functional-hierarchy hardware relationships (also reinforced by

[18] Marcelo Cataldo, James D. Herbsleb, Kathleen M. Carley, 2008, *Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity*. Proceedings of the Second ACM-IEEE Symposium on Empirical Software Engineering and Measurement, ACM, Kaiserslautern, Germany, October 9-10, pp. 2-11.

[19] For examples at Amazon and Boeing, see NRC, 2007, *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale*, Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 20, 2010.

the current revision of MIL-STD-881 on Work Breakdown Structures) that are incompatible with layered service-oriented software architectures. Research is needed on how better to reconcile these.

Goal 1.3: Facilitate Architecture-Driven Development

The core practice of architecture depends on our ability to take conceptual structures and manifest them concretely as architectural designs before systems are actually constructed. This is the essential feedback loop to reduce the most fundamental of engineering risks in innovative software engineering. As noted in Chapter 1, there is no physical limit regarding what can be accomplished at the architecture level to facilitate component-based development—in a way that addresses concerns over modularity, assurance, measurement, and other considerations.

Challenges associated with this goal are as follows:

- *Architecture designs for particular domains.* It is sometimes asserted that there are relatively few fundamental "phyla" of software, such as web services stacks, control systems of various kinds, distributed data-intensive systems, graphical user-interaction systems, etc. Within each of these phyla are various established ecosystems and also more advanced custom designs. The DoD can derive great benefits when it leads the advancement of ecosystem development for areas critical to its mission—it can directly assure attention to issues related to defense needs, rather than having to find ways to work around deficiencies in ecosystems established by others.
- *Emerging architectural concepts.* Software architecture capability continues to be enriched beyond the old model of static structural connections. Recent developments include frameworks and plug-ins, dynamic and adaptive models, service-oriented models, application frameworks, cloud and utility computing, virtualization, data-intensive models, and others. There continue to be emerging concepts that can be of benefit to complex DoD quality attribute requirements.

Goal 1.4: Facilitate Architecture Recovery

Many DoD systems do not have the benefit (and risk) of developing completely new architectures, but must find ways to provide continuity of service from legacy systems whose software is not well structured or documented (a different kind of risk). Some initial approaches for recovering service-oriented architectures for such legacy systems are emerging.[20] Further research and experience on such approaches would strengthen software producibility for the increasing number of DoD brownfield software development situations.

Area 2. Assurance: Validation, Verification, Analysis of Design and Code

Chapter 4 elaborates the significance, role, and practice of software assurance. It also identifies a number of capabilities that, if better applied and/or augmented, could greatly enhance the ability of the DoD to develop systems that are both highly capable and highly assured—and to do so with acceptable costs and programmatic risk. As noted in Chapter 1, the broadening role of systems and the consequent increase in hazards associated with very large systems combine to enhance the significance of assurance, while the challenge of assurance is increased due to the complexity of modern architectures and supply chains. On the other hand, the capacity to achieve assurance is enhanced by the recent important progress in modern programming languages, tools, modeling, and analysis capability.

[20] Two examples are the IBM VITA approach (Hopkins and Jenkins, 2008, *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*, Upper Saddle River, NJ: IBM Press) and the CMU-SEI SMART approach (Edwin J. Morris, Dennis B. Smith, and Soumya Simanta, 2008, *SMART: Analyzing the Reuse Potential of Legacy Components on a Service-Oriented Architecture Environment*, CMU/SEI-TR-2008-TN-008, Pittsburgh, PA: Carnegie Mellon University).

For new assurance technologies and practices, critical acceptance criteria must include *scalability*, which for many attributes (such as security and safety) usually also means composability, and *usability* by developers with minimal training. This greatly facilitates preventive use on a routine basis and thus enhances the ability of the DoD to structure incentives back into the supply chain for developers to create evidence along with code products. As in the case of security, many interventions in technology and practice that relate to assurance are not in the form of separate tools, but rather in the form of enhancements to tools and practices already in place for other purposes. For example, assurance considerations affect architecture modeling (e.g., to detect information paths that are not supposed to be present), requirements-related models, traceability and team information management tooling, programming language design, runtime infrastructure design, and many other areas.

Goal 2.1: Effective Evaluation for Critical Quality Attributes

This includes a wide range of technologies related to modeling, reverse engineering ("program understanding"), analysis, testing, inspection support, verification, and model checking, as well as support for managing the associated collected information and proof structures. This goal is addressed not only through the development of new techniques, but, as noted, also through the enhancement of practices and tools related to a diverse set of software engineering activities.

In general, a mature software development shop will employ multiple techniques to support assurance and evaluation. This is based on the fact that there are many different quality attributes and kinds of defects.[21] At a mature industry development organization, many different kinds of techniques and tools are used, including test frameworks, analyses with respect to different kinds of quality attributes, binary and source analysis, inspection support, metric tracking, and many others. This means that improvements in particular capabilities, when structured appropriately, can gracefully be inserted into practice.

Considerable further research is needed, however, to ensure scalability of such tradeoff analysis capabilities—that optimizing on one assurance aspect does not overly penalize other quality attributes. For example, optimizing on security has been seen to adversely affect performance (via system overhead), reliability (via single points of failure), adaptability (via recertification delays), or usability (via authentication constraints and delays), particularly for complex net-centric systems of systems.

Goal 2.2: Assurance for Components in Large Heterogeneous Systems

The goal of composable assurance for larger-scale systems is broad and complex. On the one hand, there are already a small number of composable analyses already in use (typing being a principal example). But, on the other hand, composable analyses have not yet emerged for critical security, performance, and other attributes. The pathway to such capability can include model design, theoretical and semantic research, programming language improvement (as is routinely done with major languages such as Java, Fortran, C++, C, and others), tool development, and so on. A research program that focuses on the goal would thus benefit by encompassing research approaches that address primarily a quality objective and feasibility criterion for potential adoption, but are not overly constrained with respect to specific manifestation in the process.

One of the challenges is to improve assurance for data containment in component-oriented systems. This derives from the observation (in Chapter 4) that in many large and heterogeneous software systems containing diversely sourced components (with corresponding diversity in levels of trust), the attack surface is "at the API." A particular concern is assuring that flows of data are as intended in the

[21] See comments in Chapter 4 regarding Mitre's Common Weakness Enumeration (CWE) inventory for security and code-safety attributes. There are also diverse attributes related to adaptability and flexibility, for example, modularity measures, coupling, pattern compliance, interface attributes, etc.

architectural models. This is a deeply technical challenge, as are many of the other challenges related to assurance.

Goal 2.3: Enhance the Portfolio of Preventive Methods to Achieve Assurance

In addition to the primarily evaluative techniques, interventions in development activities can greatly enhance the potential for accreting, on an ongoing basis in development, a body of evidence in support of assurance cases. For example, assurance considerations affect architecture modeling, requirements-related models, traceability and team information management tooling, programming language design, the design of runtime infrastructure, and many other areas.

If research in this area is successful, the difference it will make will be evident in two ways. First, less work will produce higher assurance, in the form of stronger claims with respect to critical quality attributes, and, second, the balancing of effort in evidence production will shift from acceptance evaluation toward development, thus reducing engineering risk with respect to assurance. A wide range of technical ideas have emerged over the years in support of this concept, and this has also influenced language design. A crude way to think about this is that an existing language, together with additional specification information (e.g., types) and analysis capability (e.g., a type checker), can lead naturally to the next-generation programming of language in which the specification information becomes intrinsic and the analysis capability is integrated with the compiler, loader, and runtime system.

Additional specific challenges include the following:

- Preventive methods also include ideas building on the concept of "proof-carrying code" or more generally "evidence-carrying code."[22]
- A significant enabler of preventive techniques in development activity is the adoption of processes and practices that enhance assurance. Examples include the Lipner/Howard Security Development Lifecycle and Gary McGraw's process.[23] These processes can continue to be enhanced and refined as new practices, tools, and languages emerge.
- Architectural building blocks can be enhanced to facilitate instrumentation and logging in systems to support real-time, near-real-time, and forensic checking of consistency with models. It is important to note that not all significant attributes can be checked in this way, although sometimes modifications to architecture can expand the scope of what can be checked dynamically.
- Develop architectures for containment such as sandboxing, process separation, virtual machines, and abstract machines. There is great opportunity to rethink basic concepts in systems software support, with a focus on achieving the simplifications that can lead to greater assurances regarding regulation of control and data flows among major components. The success of restricted ecosystems such as those evident on iPhones and other restricted platforms suggests the possibility of progress in this area.
- Employ development techniques including co-development of software, selective specifications (for functional and quality attributes), and evidence of verification (consistency) of the software code with the specifications and associated models. Different techniques apply to different properties— what may be workable for particular quality attributes may not be useful for functional, performance, or deadline properties. Most of these techniques rely on some use of explicit specifications. A goal is to reduce the extent of specification required, ultimately to fragmentary specifications that enable designers and developers to distinguish what is an intended truth from what may be an accidental truth. The intended truth may be a design commitment that can be relied upon. The accidental truth may be a consequence of a particular algorithm or infrastructure choice that needs to be subject to revision as

[22] George C. Necula and Peter Lee, 1998, *Safe, Untrusted Agents Using Proof-Carrying Code. Lecture Notes in Computer Science—Mobile Agents and Security*, London, UK: Springer-Verlag, pp. 61-91.

[23] See Michael Howard and Steve Lipner, 2006, *The Security Development Lifecycle,* Redmond, WA: Microsoft Press; also Grady McGraw, *Software Security: Building Security In,* Boston: Addison-Wesley.

technology evolves. This co-development approach is intended to facilitate incremental and iterative development practices because it simultaneously creates software and assurance-related evidence.

- The reality of enriched and diversified supply chains for software systems suggests that pervasive acceptance of preventive methods may not be fully achievable. For this reason, it is important to also address the challenge of improving a posteriori methods, including not only evaluative techniques, but also other approaches based on obfuscation and dynamic techniques.
- Develop and use programming languages that enhance assurance. The experience of software developers is that language shifts occur at unpredictable times and for unpredictable reasons. Nonetheless, these shifts are generally extensively influenced by research. For example, Ada95, Java, and C# were all influenced by the same set of ideas regarding types, safe storage management, concurrency, name space management, access management, and many other languages elements. The emerging generation of domain-specific languages and dynamic languages is now well established, providing developers with greater flexibility in development practice but also less safety than the established languages. Research work could be accelerated to augment these languages with features that preserve the usability and flexibility while enhancing the potential for assurance.

Area 3. Process Support and Economic Models for Assurance and Adaptability

Chapters 2 and 4 both address issues related to process and assurance and suggest the following as research goals.

Goal 3.1: Enhance Process Support for Both Agile and Assured Software Development

This includes both product and process architectures based on identifying the parts of the product and process most needing agility or assurance, and organizing the architectures around them. For products, one way to do this is by encapsulating the major sources of change into modules to be handled by agile methods.[24] Examples of such sources of change are user interfaces, interfaces to independently evolving systems, or device drivers. For projects, one way to do this is to partition evolutionary development around stabilized high-assurance development increments, while a parallel team is handling the change traffic and developing the specifications and plans for the next increment.

It also includes further improvements in information management for teams and larger development organizations. Areas of focus could beneficially include improved traceability particularly for formal and "semi-formal" information items, integration of models and analyses and simulation, and measurement support to facilitate iteration and evaluation (e.g., to dynamically identify and adapt to new sources of rapid change).

Goal 3.2: Address Supply-Chain Challenges and Opportunities

As supply chains are enriched and diversified, there is an increasing potential benefit from tools that can manage a joint corpus of information and whose content and sharing is regulated according to a contractual relationship. Enhancements of this kind can better support evidence production by producers to accelerate client acceptance evaluation. The enhancements can also better support intertwined iterations. Such tools need to be reinforced by contractual provisions enabling visibility and measurability of development and risk management plans and progress vs. plans, both along a supply chain, and up and down the subcontractor chains.

[24] David Parnas, 1978, "Designing Software for Ease of Extension and Contraction," *Proceedings of the 3rd International Conference on Software Engineering*, IEEE, Atlanta, GA, May 10-12, pp. 264-277.

Goal 3.3: Facilitate Application of Economic Principles to Decision Making

An additional area of potential significance is the development of applicable economic models to guide decision making, for example, related to the interplay of architecture choices, component and ecosystems choices, supply-chain choices, and attributes of cost, risk, schedule, and quality. As discussed in Chapter 2, prioritizing features to enable time-certain development provides a strong proxy for economic value-based management and decision making.

Goal 3.4: Develop and Apply Policy Guidance and Infrastructure for Conducting Evidence-Based DoD Milestone Reviews

As also discussed in Chapter 2, this task includes establishing the evidence of solution feasibility as a first-class deliverable, reviewing evidence-development plans, and tracking evidence development progress vs. plans via earned value management. It also requires research into which classes of process-focused evidence development (models, simulations, prototypes, benchmarks, exercises, instrumentation, etc.) are best suited for which classes of system elements.

Goal 3.5: Enhance Process Support for Integrated Definition and Development of System Hardware, Software, and Human Factors Requirements and Architectural Solutions

Too often, system architectures are driven by hardware relationships that overly constrain software and human factors solutions. Examples of approaches are "soft systems engineering," systems architecting, co-evolution, incremental iterative development (IID) models based on spiral development, and Brooks's design processes and patterns.[25]

Area 4. Requirements

The challenges for requirements are, in many respects, similar to those of architecture. How to achieve early validation? How to express the information that is gathered from stakeholders concerning both functional requirements and quality attributes? How to achieve traceability and model consistency that effectively links requirements with architecture and assurance?

As noted in the previous chapters, requirements are only occasionally fully established at the outset of the development of an innovative software system. More often, there are early constraints on quality attributes, definitions of the overall scope of function and interlinking, and a few other "shall" or "must-have" constraints. Many of the other elements that eventually become manifest as features or quality attributes are in fact the result of early iterations with stakeholders, and many of these are informed by the improved understanding of both the technological and operational environments as they evolve. In other words, requirements engineering is an ongoing activity throughout development. For long-lived systems, as noted in the 2006 Software Engineering Institute (SEI) report *Ultra-Large-Scale Systems*, requirements engineering is ongoing throughout the lifetime of the system.

[25] Soft systems engineering (see Peter Checkland, 1981, *Systems Thinking, Systems Practice*, Hoboken, NJ: Wiley); systems architecting (see Eberhardt Rechtin, 1991, *Systems Architecting: Creating & Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall), co-evolution (see Mary Lou Maher, Josiah Poon, and Sylvie Boulanger, 1996, "Formalizing Design Exploration as Co-evolution: A Combined Gene Approach," pp. 3-30 in *Advances in Formal Design Methods for CAD*, John S. Gero and Fay Sudweeks, eds., London, UK: Chapman and Hall), the incremental commitment model upgrade of spiral development (NRC, Richard W. Pew and Anne S. Mavor, eds., 2007, *Human-System Integration in the System Development Process: A New Look*, Washington, DC: National Academies Press, available online at http://books.nap.edu/catalog.php?record_id=11893), and Brooks's design processes and patterns (see Fred Brooks, 2010, *The Design of Design: Essays from a Computer Scientist*, New York, NY: Addison-Wesley).

Goal 4.1: Expressive Models and Supporting Tools

A feature of modern requirements methodology is the capture of scenarios and use cases and the expression of these using effective but mostly informal notations. For agile or feature-driven developments, attention is addressed to the granularity of featuring, so to speak, because that becomes both the basis of priority setting ("above the line") and the metric of progress ("velocity" or "burn down"). For innovative large systems, there is more focus on capturing the model in a sufficiently precise form to support progress measurement and acceptance evaluation. Regardless of the approach, however, there are common core technical challenges, which are to improve our ability to express requirements-related models (in the sense of unified modeling language (UML) scenarios and use cases), to reason about those models (in the sense of the Massachusetts Institute of Technology's Alloy), and to facilitate traceability with respect to architecture and implementation (correspondence measures). Requirements engineering is fundamentally about the transition from informal human expression to explicit structured representations. Any incremental improvement in formality that doesn't compromise overall expressiveness or efficiency of operation has the potential to make a big difference with respect to these goals.

Related to this goal is the development of improved domain-specific models and methods that pertain to critical defense domains such as control systems, command and control, large-scale information management, and many others.

Goal 4.2: Support Traceability and Early Validation

Traceability is more readily achieved when the feature-driven model is adopted, but this is not always readily applicable to defense systems. Research on requirements expression will result in improvements to models, tooling, and early validation practices (e.g., prototyping and simulation). As part of this effort, it is essential to also address traceability issues, because these have a profound influence on assurance and validation generally.

Goal 4.3: Process Support for Stakeholder Engagement and Model Development

Stakeholders in large projects may come from diverse perspectives and may have diverse interests. The requirements can often appear to be a negotiation among stakeholders regarding the significance of various functional features and quality attributes. This creates a challenge of avoiding both over-commitment (e.g., through negotiation) to particular characteristics as well as under-commitment. What modeling mechanisms, processes, and tools can be developed to assist stakeholders in identifying goals and models, and in managing not just *what* is committed to, but also *how much* commitment is made? This is particularly critical in incremental and iterative development projects.

Area 5. Language, Modeling, Coding, and Tools

As noted in the previous chapters, programming languages and associated capabilities have a considerable influence on the major factors identified in this report—architecture, assurance, process, and measurement. For example, programming language improvements have influence on the ability of architects to achieve goals related to system structure and modularity. More generally, programming languages are the medium by which human developers convey their intent both to the computer and to other developers. As such, a programming language both constrains what a developer can say and at the same time encourages particular styles of expression. As noted in the previous chapters, programming language design has considerable influence on the major factors identified in this report—architecture, assurance, process, and measurement. For example, programming language improvements have influence on the ability of architects to enforce goals related to system structure and modularity. Modularity is much more than a matter of control and data flow. There are abstractions related to objects, types, and

actions that are increasingly supported in modern languages, and these enable developers to express domain concepts more directly in program text. They also enable architects to render their abstract models for system modular structures more directly into the explicit structure of programs.

Modularity is much more than a matter of control and data flow; there are abstractions related to objects, types, and actions that are increasingly supported in modern languages, and these not only enable developers to express domain concepts more directly in program text[26] but also enable architects to render their abstract models for system modular structures more directly into the explicit structure of programs.

Goal 5.1: Enhance the Expressiveness of Programming Language to Address Current and Emerging Challenges

Despite many declarations of success, the evolution of programming languages continues, and it is driven by strong demand for improvements from developers seeking greater levels of expressiveness (e.g., through direct expression of concepts such as higher-order functions,[27] deterministic parallelism, atomicity, data permissions, and so on), improved ability to support particular domains (either through direct expression as intrinsics or through the ability to provision appropriate libraries and frameworks), improved flexibility for developers (e.g., dynamic languages that compromise less static checking for a more rapid iterative development model, but with more risk of unwanted runtime errors), improved a priori assurance (e.g., through the simultaneous development of code, specifications, and associated proofs for assurance), and improved access to scalable performance (e.g., through intrinsics such as Generate/MapReduce that support data-intensive computing in microprocessor clusters).

Goal 5.2: Enhance Ability to Exploit Modern Concurrency, Including Shared Memory Multicore and Scalable Distributed Memory

For the past 30 years, as a consequence of the steady improvements in processor design, software developers have been given a doubling in performance every year and a half, adding up to a million-fold improvement in three decades. Over that period, the same code ran faster on the new chips. In the past few years, processor clock speeds have topped out; there are now multiple processors on a chip, and chip designers continue to provide the expected performance improvement, but only in a potential way and accessible only to those software developers who can harness the power of multiple processors. Suddenly, everything has to be "done by committee"—by multiple threads of control coordinating together to get the work done. It is said that Moore's Law has given way to Amdahl's Law. To make matters more difficult, the ability of multiple threads to access shared state in memory does not scale

[26]In the early days of Fortran, for example, the only data types in the language were numbers and arrays of various dimensionalities. Any program that manipulated textual data, for example, needed to encode the text characters, textual strings, and any overarching paragraph and document structure very explicitly into numbers and arrays. A person reading program text would see only numerical and array operations, because that was the limit of what could be expressed in the notation. This meant that programmers needed to keep track, in their heads or in documentation, of the nature of this representational encoding. It also meant that testers and evaluators needed to assess programs through this (hopefully) same layer of interpretation. With modern languages (including more modern Fortran versions), these structures can be much more directly expressed—characters and strings are intrinsic in nearly all modern languages. This is a simple illustrative example, but the point remains: There are concepts and structures in domains significant to defense that, in modern languages, must be addressed through similar representational machinations. This is a part of the "endless value spiral" argument of Chapter 1, and it explains why we should not expect any plateau in the evolution of programming languages, models, and other problem-relevant expressive notations. Indeed, it is why language names such as "Fortran" and "Ada" have the staying power of strong brands, even when the specific languages to which they refer are evolving quite rapidly (for example, Ada83 to Ada95 and thence to Ada 2005).

[27]An example is Microsoft's F#, which builds on two decades of work on advanced functional languages such as Standard ML and Haskell. Another example is Sun's Fortress language, which builds on a combined heritage of functional programming, deterministic parallelism, and numerical computation.

up—it must eventually be supplanted by distributed models, with information shared using message passing. This hybrid approach, combined with a distributed approach to scalable storage, is the reality of many modern high-performance data centers.[28]

Software developers, language designers, and tool developers are still struggling to figure out how to harness the concurrency in a way that works well for software development. What are the correct abstractions? What are suitable concepts for data structures? How can assurance be achieved when programs operate in non-deterministic fashion? This provisioning of modern computing power is a major challenge for language designers and tool designers.

Goal 5.3: Enhance Developer Productivity for New Development and Evolution

As noted above, languages enhanced with models and tools often merge into new languages that incorporate the model concepts directly in the language design. But there is a growing suite of tool capabilities that are conceptually separate from language, and the delivery of these capabilities is a significant influence on developer and team productivity and on software producibility generally. Modern tools such as the open source Eclipse (created by IBM[29]) and Microsoft's Visual Studio environment for "managed code" provide rich features to support application development generally. They also have tailored support for development within certain ecosystems, such as the Visual Studio support for web applications developed within the Microsoft Asp.NET framework. Individual developer tools are often linked with team capabilities, which include configuration management of code and related artifacts, defect and issue tracking and linking, build and test support, and management of team measures and processes. This linkage greatly empowers small teams and, increasingly, larger development organizations.

Area 6. Cyber-Physical Systems

DoD systems are increasingly operating in large-scale, network-centric configurations that take input from many remote sensors and provide geographically dispersed operators with the ability to interact with the collected information and to control remote effectors. In circumstances where the presence of humans in the loop is too expensive or their responses are too slow, these so-called cyber-physical systems must respond autonomously and flexibly to both anticipated and unanticipated combinations of events during execution. Moreover, cyber-physical systems are increasingly being networked to form long-lived systems of systems—and even ultra-large-scale systems[30]—that must run unobtrusively and largely autonomously, shielding operators from unnecessary details (but keeping them apprised so they can react during emergencies), while simultaneously communicating and responding to mission-critical information at heretofore infeasible rates.

Cyber-physical systems are increasingly critical in defense applications of all kinds and at all levels of scale, including distributed resource management in shipboard defense systems, coordinating groups of unmanned air vehicles, and controlling low-power sensors in tactical urban environments. These are systems with very close linkage of hardware sensors and effectors with software control. They are often structured as control systems, but also can involve multiple complex interacting control systems, such as in deconflicting multiple call-for-fire requests in a crowded battlespace consisting of joint services and coalition partners.

One critical area of concern is the creation and validation of the cyber-physical stack. For example,

[28]These issues are the focus of a forthcoming report from the National Research Council, *The Future of Computing Performance: Game Over or Next Level?*, Samuel Fuller and Lynette Millett, eds., Washington, DC: National Academies Press, forthcoming.

[29] Siobhan O'Mahony, Fernando Cela Diaz, and Evan Mamas, 2005, "IBM and Eclipse (A)," Harvard Business School Case 906007, Cambridge, MA: Harvard University Press.

[30] Software Engineering Institute, 2006, *Ultra-Large-Scale Systems: The Software Challenge of the Future*, Pittsburgh, PA: Carnegie Mellon University. Available online at http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf. Last accessed August 20, 2010.

how to evolve the development of distributed real-time and/or embedded systems from a cottage craft that does not generally yield scalable or readily assurance solutions to a more robust approach guided by model-integrated computing, domain-specific languages and analysis tools, and control-theoretic adaptation techniques. This is a significant challenge for language and platform design, ecosystem development, tool design, and practices.[31]

Another challenge facing the DoD is how to routinize and automate more of the development of embedded cyber-physical control systems. There are particular challenges related to scalability, predictability, and evolvability of systems. Assurance is also a major issue, and it is exacerbated by the lack of ability to reliably connect models with code and with the various components in the current generation of embedded software stack, which are typically optimized for time/space utilization and predictability, rather than ease of understanding, analysis, composition, scalability, and validation.

Yet another challenge facing DoD cyber-physical systems—particularly net-centric cyber-physical systems—is how to handle variability and control adaptively and robustly. Cyber-physical systems today often work well as long as they receive all the resources for which they were designed in a timely fashion, but fail completely under the slightest anomaly. There is little flexibility in their behavior, that is, most of the adaptation is pushed to end users or administrators. Instead of hard failure or indefinite waiting, what net-centric cyber-physical systems require is either *reconfiguration* to reacquire the needed resources automatically or *graceful degradation* if they are not available.

Goal 6.1: Accelerate Ecosystem Development for Cyber-Physical Systems

Today, it is too often the case that substantial effort expended to develop cyber-physical systems focuses on either (1) building ad hoc solutions based on tedious and error-prone low-level platforms and tools or (2) cobbling together functionality missing in off-the-shelf real-time and embedded operating systems and middleware. As a result, subsequent composition and validation of these ad hoc capabilities is either infeasible or prohibitively expensive. One reason why redevelopment persists is that it is still often relatively easy to pull together a minimalist ad hoc solution, which remains largely invisible to all except the developers and testers. Unfortunately, this approach yields brittle, error-prone systems and substantial recurring downstream ownership costs, particularly for complex and long-lived network-centric DoD systems and larger-scale systems-of-systems.

One of the most immediate goals is therefore to accelerate ecosystem development for cyber-physical systems. There has been considerable exploration of this area in a multi-agency setting under the auspices of the NITRD coordination activity (see Box 1.5), and there are benefits to linking it with other efforts related to software producibility. There are opportunities to exploit and advance modern language concepts, innovative operating system and middleware ideas, scheduling and resource management techniques, and code generation capabilities.

Achieving this goal will require new cyber-physical system software architectures whose component functional and quality-of-service (QoS) properties can be expressed with sufficient precision (e.g., via the use of model-integrated computing techniques and domain-specific languages and tools) that they can be predictably assembled with each other, leaving less lower-level complexity for application developers to address and thereby reducing system development and ownership costs. In particular, cyber-physical system ecosystems must not simply build better device drivers, operating system schedulers,

[31]This challenge was discussed in the committee's 2007 workshop report. See NRC, 2007, *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale,* Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11936. Last accessed August 20, 2010. It has also been explored in the NRC report *Software for Dependable Systems*: *Sufficient Evidence?* See NRC, Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds., 2007, *Software for Dependable Systems, Sufficient Evidence?* Washington, DC: National Academies Press. Available online at http://www.nap.edu/catalog.php?record_id=11923. Last accessed August 20, 2010. It has been the subject of a series of workshops under the auspices of the NITRD HCSS area sponsored by NSF and other agencies. For more information see http://www.nitrd.gov/subcommittee/hcss.aspx. Last accessed August 20, 2010.

or middleware brokers in isolation, but rather integrate these capabilities together and deliver them to applications in ways that enable them to achieve fine-grained tradeoffs between key QoS properties, such as throughout, latency, jitter, scalability, security, and reliability.

Key R&D breakthroughs needed to meet the goal of accelerating ecosystem development for cyber-physical systems involve devising new languages and platforms that enable users and operators to clearly understand the QoS requirements and usage patterns of software components so it becomes possible to analyze whether or not these requirements are being (or even can be) met and to aggregate these requirements, making it possible to form decisions, policies, and mechanisms that can support effective global management in net-centric environments. Meeting these needs will require flexibility on the parts of both the application components and the cyber-physical system infrastructure ecosystem used throughout DoD systems.

Goal 6.2: Develop Architectures and Software Frameworks to Support Embedded Applications

Embedded cyber-physical systems can operate robustly in harsh environments through careful coordination of a complex network of sensors and effectors. Given the increasing complexity of emerging DoD embedded cyber-physical systems, such fine-tuned coordination is ordinarily a nearly impossible task, both conceptually and as a software engineering undertaking. Model-based software development uses models of a system to capture and track system requirements, automatically generate code, and semi-automatically provide tests or proofs of correctness. Models can also be used to build validation proofs or test suites for the generated code.

Model-based software development removes much of the need for fine-tuned coordination, by allowing programmers to read and set the evolution of abstract state variables hidden within the physical system. For example, a program might state, "produce 10.3 seconds of 35% thrust," rather than specify the details of actuating and sensing the hardware (e.g., "signal controller 1 to open valve 12," and "check pressure and acceleration to confirm that valve 12 is open"). Hence a model-based program constitutes a high-level specification of intended state evolutions. To execute a model-based program an interpreter could use a model of a controlled plant to continuously deduce the plant's state from observations and to generate control actions that move the plant to specified states.

Achieving the goal of model-based embedded software development requires new expressive languages for specifying intended state evolutions and plant behavior, automated execution methods for performing all aspects of fine-grained coordination, and software architectures and frameworks for pervasive/immersive sensor networks. Key R&D breakthroughs needed to meet the goal of developing architectures and software frameworks to support embedded applications include closing the consistency gap between model and code, preserving structural design features in code, translating informal requirements into formal requirements, tracing requirements into implementation, integrating disparately modeled submodels, and enriching formalisms that support QoS properties, as well as techniques that support rapid reconfiguration and reliability with unreliable components.

Goal 6.3: Develop and Validate Technologies That Support Both Variability and Control in Net-Centric Cyber-Physical Systems

As DoD cyber-physical systems become increasingly interconnected to form net-centric systems of systems it is becoming clear that (1) different levels of service are possible and desirable under different conditions and costs and (2) the level of service in one property must be coordinated with and/or traded off against the level of service in others to achieve the intended mission results. To date, little work has focused on techniques for controlling and trading off the overall behavior of these integrated net-centric cyber-physical systems. Another key goal is therefore to develop and validate new technologies that support both variability and control in net-centric cyber-physical systems.

Achieving this goal will require devising new adaptive and reflective software technologies, recog-

nizing that not all requirements can be met all of the time, yet still ensuring predictable and controllable end-to-end behavior. In adaptive software technologies, the functional and QoS-related properties of cyber-physical software can be modified either statically (e.g., to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies) or dynamically (e.g., to optimize system responses to changing environments or requirements, such as changing component interconnections, power levels, CPU/network bandwidth, latency/jitter, and dependability needs).

Reflective software technologies go further to permit automated examination of the capabilities they offer and automated adjustment to optimize those capabilities. Reflective techniques make the internal organization of systems—as well as the mechanisms used in their construction—both visible and manipulable for application and infrastructure programs to inspect and modify at runtime. Reflective technologies thus support more advanced adaptive behavior and more dynamic strategies keyed to current circumstances, that is, necessary software adaptations can be performed autonomously based on conditions within the system, in the system's environment, or in system QoS policies defined by operators.

Key R&D breakthroughs needed to meet the goal of developing and validating adaptive and reflective software for net-centric cyber-physical systems involve investigating ways to make such modifications dependably (e.g., while meeting stringent—often conflicting—end-to-end QoS requirements) while simultaneously ensuring that the system functional requirements are met.

Area 7. Human-Systems Integration

It is significant that most large-scale complex enterprise systems include fallible humans as constituent elements, but there has been a lack of design practices, including architecture concepts and development processes, that account for the ways in which humans integrate into systems as participants. Human-systems integration (HSI) is about much more than the colors of pixels and the design of graphical user integration frameworks. The presence of humans in a system, such as pilots in an airplane, fundamentally affects the design and architecture of that system.

This issue was the subject of a separate NRC report[32] and is not elaborated upon here except to emphasize some of its software-related recommendations:

- Conduct a research program with the goal of revolutionizing the role of end users in designing the system they will use.
- Conduct research to understand the factors that contribute to system resilience, the role of people in resilient systems, and how to design more resilient systems.
- Refine and coordinate the definition of a systems development process that concurrently engineers the system's hardware, software, and human factors aspects, and accommodates the emergence of HSI requirements, such as the incremental commitment model.
- Research and develop shared representations to facilitate communication across different disciplines and lifecycle phases.
- Research and develop improved methods and testbeds for systems-of-systems HSI.
- Research and develop improved methods and tools for integrating incompatible legacy and external-system user interfaces.

[32] See NRC, 2007, *Human-System Integration in the System Development Process: A New Look*, Washington, DC: National Academies Press, Available online at http://www.nap.edu/openbook.php?record_id=11893. Last accessed August 20, 2010.

Summary Findings and Recommendations

Finding 5-2: Technology has a significant role in enabling modern incremental and iterative software development practices at levels of scale ranging from small teams to large distributed development organizations.

Recommendation 5-1: The DoD should take immediate action to reinvigorate its investment in software producibility research. This investment should be undertaken through a diversity of research programs across the DoD and should include academia, industry labs, and collaborations.

Recommendation 5-2: The DoD should take action to undertake DoD-sponsored research programs in the following areas identified as critical to the advancement of defense software producibility: (1) architecture modeling and architectural analysis; (2) assurance: validation, verification, analysis of design and code; (3) process support and economic models for assurance and adaptability; (4) requirements; (5) language, modeling, coding, and tools; (6) cyber-physical systems; and (7) human-systems integration.

Appendixes

Appendix A

Briefers to the Committee

SEPTEMBER 27-28, 2006, FIRST COMMITTEE MEETING

Annmarie Bunts, Lockheed Martin Systems Integration
Grady Campbell, Software Engineering Institute
Larry Druffel, SCRA (retired)
Robert Gold, Office of the Deputy Under Secretary for Science and Technology/Information Systems
Ronald T. Kadish, Booz Allen Hamilton
James Larus, Microsoft Research
Robert Nesbit, MITRE Center for Integrated Intelligence Systems
Linda Northrop, Software Engineering Institute
Tom Rodgers, Lockheed Martin
Walker Royce, IBM
Andre van Tilborg, Office of the Deputy Under Secretary for Science and Technology/Information Systems

JANUARY 17, 2007, WORKSHOP ON UNCERTAINTY AT SCALE

Cynthia Andres, Three Rivers Institute
Kristen J. Baldwin, Office of the Under Secretary of Defense, Acquisition, Technology, and Logistics
Kent Beck, Three Rivers Institute
Kris Britton, National Security Agency, Center for Assured Software
Mary Ann Davidson, Oracle Corporation
Joe Jarzombek, Department of Homeland Security
Patrick Lardieri, Lockheed Martin
Gary McGraw, Cigital, Inc.
Richard W. Selby, Northrop Grumman
Alfred Spector,[1] Google, Inc.

[1]As of November 2007. At the time of the workshop, Dr. Spector was an independent consultant.

John Vu, Boeing
Werner Vogels, Amazon.com

APRIL 24, 2007, THIRD COMMITTEE MEETING

Thomas Blann, Office of the Director of Operational Test and Evaluation

SEPTEMBER 16-17, 2008, FOURTH COMMITTEE MEETING

Kristen Baldwin, Acting Director, Systems and Software Engineering, ODUSD(A&T)
Dan Reed, Director of Scalable and Multicore Computing Strategy, Microsoft

Appendix B

Biosketches of Members of the Committee

William L. Scherlis, *Chair*, is a professor in the School of Computer Science at Carnegie Mellon University (CMU). He is the founding director of CMU's PhD program in software engineering and director of CMU's Institute for Software Research (ISR). His research relates to software assurance, software evolution, and technology to support software teams. Dr. Scherlis joined the CMU faculty after completing a PhD in computer science at Stanford University, a year at the University of Edinburgh (Scotland) as a John Knox Fellow, and an A.B. at Harvard University. He was the lead principal investigator of the 4-year High Dependability Computing Project (HDCP), in which CMU led a collaboration of five universities to help NASA address long-term software dependability challenges. Dr. Scherlis is involved in a number of activities related to technology and policy, recently testifying before Congress on innovation and information technology, and, previously, on roles for a federal Chief Information Officer (CIO). He interrupted his career at CMU to serve at the Defense Advanced Research Projects Agency (DARPA) for 6 years, departing in 1993 as senior executive responsible for coordination of software research. While at DARPA he had responsibility for research and strategy in computer security, aspects of high-performance computing, information infrastructure, and other topics. Dr. Scherlis was a member of the National Research Council (NRC) study committee on cybersecurity and the DARPA Information Science and Technology Study Group (ISAT). He recently completed chairing an NRC study on information technology, innovation, and e-government. He has led or participated in national studies related to cybersecurity, crisis response, analyst information management, Department of Defense software management, and health care informatics infrastructure. He has been an advisor to major information technology (IT) companies. He has served as program chair for a number of technical conferences, including the ACM Foundations of Software Engineering (FSE) Symposium. He has more than 75 scientific publications.

Robert F. Behler is a senior vice president and general manager in the MITRE Corporation Command and Control Center. The center serves MITRE's Department of Defense sponsors and focuses on creating a joint command, control, and communications system. Mr. Behler leads the center's work for Department of Defense sponsors. Before joining MITRE in April 2006, Mr. Behler was general manager of Precision Engagement at Johns Hopkins University's Applied Physics Laboratory. In this position he supervised more than 250 scientists and engineers working on advanced command, control, intelligence, surveillance, and reconnaissance (C2ISR) programs for the Department of Defense. Under Mr.

Behler's leadership, the Precision Engagement organization turned new and emerging technologies into transformational operational capabilities. Mr. Behler retired from the Air Force as a major general in 2003. During his distinguished 31-year career, he accumulated extensive experience managing and developing advanced command, control, communications, computers, intelligence, surveillance, and reconnaissance (C4ISR) technologies at all levels. Before retiring, Mr. Behler was commander of the Air Force C2ISR Center at Langley Air Force Base, where he was principal C2ISR advisor to the secretary and chief of staff of the Air Force. Prior to that, he served as deputy commander of NATO Joint Headquarters North in Stavanger, Norway, and was the senior U.S. military officer in Scandinavia. He has also served as director of command, control, communication, computers, and intelligence at the U.S. Strategic Command at Offutt Air Force Base and as chief of the U.S. Air Force Senate Liaison Office. Mr. Behler entered the Air Force in 1972 as a distinguished graduate of the Air Force Reserve Officer Training Corps program at the University of Oklahoma. Mr. Behler received his bachelor's and master's degrees in aerospace engineering from the University of Oklahoma in 1970 and 1972, respectively. He is a graduate of the U.S. Air Force Test Pilot School at Edwards Air Force Base and was a National Security Fellow at Harvard University's John F. Kennedy School of Government in 1990. He received a master's degree in business administration from Marymount University in 1991. He is an associate fellow of the Society of Experimental Test Pilots and a member of the Armed Forces Communications and Electronics Association.

Barry W. Boehm is TRW Professor of Software Engineering and founding Director Emeritus of the Center for Systems and Software Engineering at the University of Southern California. He is also director of research of the DoD-Stevens-USC Systems Engineering Research Center and co-director of the USC-Chinese Academy of Sciences Joint Laboratory for Software Engineering. His contributions to the field include the Constructive Cost Model (COCOMO); the Spiral Model of the software process; the Theory W (win-win) approach to and tools for software management and requirements determination. Between 1989 and 1992, he served within the U.S. Department of Defense as director of the DARPA Information Science and Technology Office and as director of the DDR&E Software and Computer Technology Office. He worked at TRW from 1973 to 1989, culminating as chief scientist of the Defense Systems Group, and at the Rand Corporation from 1959 to 1973, culminating as head of the Information Sciences Department. He was a programmer-analyst at General Dynamics between 1955 and 1959. He has served on the board of several scientific journals, including the *IEEE Transactions on Software Engineering*, *IEEE Computer*, *IEEE Software*, *ACM Computing Reviews*, *Automated Software Engineering*, *Software Process*, and *Information and Software Technology*. He has served as chair of the AIAA Technical Committee on Computer Systems, chair of the IEEE Technical Committee on Software Engineering, and as a member of the Governing Board of the IEEE Computer Society. He has served as chair of the Air Force Scientific Advisory Board's Information Technology Panel and chair of the Board of Visitors for the CMU Software Engineering Institute. He is a fellow of the leading professional societies in computing (ACM), aerospace (AIAA), electronics (IEEE), and systems engineering (INCOSE), and a member of the National Academy of Engineering. Dr. Boehm received his B.A. degree from Harvard in 1957; his M.S. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, all in mathematics; and an honorary SC.D. degree from the University of Massachusetts in 2000 in computer science.

Lori A. Clarke is a professor of computer science at the University of Massachusetts, Amherst, and co-director of the Laboratory for Advanced Software Engineering Research. She is an ACM Fellow and a board member of CRA-W. She is a former IEEE Distinguished Visitor, ACM National Lecturer, IEEE Publication Board member, associate editor of *ACM TOPLAS* and *IEEE TSE*, member of the CCR NSF advisory board, ACM SIGSOFT secretary/treasurer, vice-chair and chair, vice-chair of CRA, and co-chair of CRA-W, as well as a 1990 recipient of the University of Massachusetts Chancellor's Medal and a 1993 recipient of a University Faculty Fellowship. Dr. Clarke has worked in the area of software engineering, particularly on software analysis and testing for many years. She was one of the primary developers of symbolic execution, a technique used to reason about the behavior of software systems and

for selecting test data, and she has made contributions in the areas of software architecture and object management. Recently her work has focused on analysis of concurrent systems. With colleagues, she developed FLAVERS, a static analysis tool that uses data-flow analysis to verify user-specified properties of concurrent systems, and PROPEL, a system that complements FLAVERS and other model checking systems by helping users elucidate the details of the properties to be proven. She received her B.A. in mathematics (1969) from the University of Rochester and her Ph.D. degree in computer science (1976) from the University of Colorado.

Michael A. Cusumano is the Sloan Management Review Distinguished Professor of Management at the Massachusetts Institute of Technology's Sloan School of Management, with a joint appointment in MIT's Engineering Systems Division. He received a B.A. degree from Princeton in 1976 and a Ph.D. from Harvard in 1984, and completed a postdoctoral fellowship in production and operations management at the Harvard Business School during 1984-1986. He has received two Fulbright Fellowships as well as a Japan Foundation Fellowship for study at the University of Tokyo. He is currently a director of Patni Computer Systems, one of the largest IT services and custom software development firms based in India (NYSE: PTI), and Eliza Corporation, a specialist in speech recognition software applications, focused on healthcare. He is on the advisory board of FixStars Corp., a Japanese developer of high-performance computing applications. Professor Cusumano was named one of the most influential people in technology and IT by Silicon.com in 2009. He has consulted for approximately 100 firms and organizations around the world and is the author or co-author of 9 books. His newest book, *Staying Power: Six Enduring Principles for Managing Strategy and Innovation in an Uncertain World* (2010), is based on the 2009 Clarendon Lectures in Management Studies at Oxford University. *The Software Business* (2004) was named one of the top business books of the year by Steve Lohr of the *New York Times*. The international best-seller *Microsoft Secrets* (1995, with Richard Selby) has been translated into 14 languages. *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft* (1998, with David Yoffie) was named a top-10 book of the year by *Business Week*. In addition, he has published *Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation* (2002, with Annabelle Gawer); *Thinking Beyond Lean: Multi-Project Management at Toyota and Other Companies* (1998, with Kentaro Nobeoka); *Strategic Thinking for the Next Economy* (2001, with Costas Markides); *Japan's Software Factories* (1991); and *The Japanese Automobile Industry* (1985).

Mary Ann Davidson is the chief security officer at Oracle Corporation, responsible for Oracle product security, as well as security evaluations, assessments, and incident handling. She represents Oracle on the board of directors of the Information Technology Information Security Analysis Center (IT-ISAC), and the editorial advisory board of SC Magazine. She was named one of Information Security's top five "Women of Vision" and is a 2004 Fed100 award recipient from *Federal Computer Week*. She has served on the Defense Science Board and is a member of the Center for Strategic and International Studies Cyber Commission for the 44th President. She was recently named to the Information Systems Security Association Hall of Fame. She has also testified on the issue of cybersecurity to the U.S. House of Representatives (Energy and Commerce Committee, Armed Services Committee, and Homeland Security Subcommittee on Cybersecurity, Emerging Threats and Science and Technology) and the U.S. Senate (Commerce, Science and Technology Committee). Ms. Davidson has a B.S.M.E. from the University of Virginia and an M.B.A. from the Wharton School of the University of Pennsylvania. She has also served as a commissioned officer in the U.S. Navy Civil Engineer Corps, during which she was awarded the Navy Achievement Medal.

Larry Druffel is director emeritus and visiting scientist at the Software Engineering Institute at Carnegie Mellon University, where he was the director from 1986 to 1996. From 1996 to 2006, he was president and CEO of SCRA, a public, non-profit research and development corporation engaged in the application of advanced technology. He is a member of the board of directors of Teknowledge Corporation. He was vice president for business development at Rational Software from 1983 to 1986 and served on the board of directors of Rational from 1986 to 1995. Dr. Druffel was on the faculty at the USAF Academy.

He later managed research programs in advanced software technology at DARPA. He was founding director of the Ada Joint Program Office, and then served as director of Computer Systems and Software (Research and Advanced Technology) in the Office of the Secretary of Defense. He is the co-author of a computer science textbook and over 35 professional papers, including the chapter titled "Information Warfare" for the ACM Fiftieth Anniversary Book *Beyond Computing*. He has a B.S. in electrical engineering from the University of Illinois, an M.Sc. in computer science from the University of London, and a Ph.D. in computer science from Vanderbilt University. Dr. Druffel is a fellow of the IEEE, and a fellow of the ACM. He has served on engineering advisory boards of the University of South Carolina, Clemson, and Embry Riddle University. Dr. Druffel chaired the AF Science Advisory Board Study on Information Architecture and co-chaired the Defense Science Board study on acquiring defense software commercially. He led the Defensive Information Warfare Panel for the AFSAB "New World Vistas." He has served on numerous AFSAB, DSB, and NRC studies dealing with the use of information technology, including the National Research Council study *Engineering Challenges to the Long-Term Operation of the International Space Station*.

Russell Frew is the vice president, CTO in the $17 billion Lockheed Martin, Electronic Systems Business Unit (ES). In this capacity he oversees both technology development and the program performance of over 18,000 engineers and 1400 programs. He is frequently called upon to lead engineering assistance teams that engage major programs across the corporation. In his capacity as the chief technical officer, he is also responsible for technology strategy and the investment plan. Additionally, Mr. Frew has executive responsibility for the LM Advanced Technology Laboratories in Cherry Hill, NJ. From 1999 to late 2003, Mr. Frew was on special assignment for the corporation. His duties made him the focal point between ES and Aeronautics business areas. In this capacity he led major program teams working issues on the F-22's avionics, the F-35 Joint Strike Fighter's Mission System, and the F-16 Advanced Mission Computer. As part of the COTS revolution, Mr. Frew authored and led the Lockheed Martin Proven Path electronics program. Originally conceived as an LM strategy for JSF, Proven Path evolved into an engineering discipline now being widely applied across fighter aircraft, Army missiles, and Navy ships. Prior to his appointment as vice president, Advanced Technology for MS2 in 1999, Frew spent 18 months as vice president, Technology for Government Electronics Systems (GES) in Moorestown, NJ. While with GES he managed technology programs such as COMBATS, which successfully introduced modular, object-oriented software for modern ship combat systems. From June 1996 to March 1997, Frew was the managing director of the Lockheed Martin Corp. Advanced Technology Laboratories—an applied research facility that develops advanced technology hardware and software solutions for varied defense applications. Prior to 1996, Mr. Frew managed the General Electric Aerospace, Artificial Intelligence lab for 8 years. There he oversaw activities in anti-submarine warfare, attack helicopter sensor-based reasoning, expert systems, and real-time embedded architectures. He succeeded in getting the *Sea Shadow* prototype stealth ship operational and then used this platform to test numerous technology concepts in a fully operational environment at sea with the U.S. Navy. In 1985 the Defense Advanced Research Projects Agency (DARPA) selected him as one of the original program managers on the national Strategic Computing program DARPA initiated to meet Japan's Fifth Generation challenge. Mr. Frew holds graduate and undergraduate degrees. He has served as a study panel member at the National Academy of Sciences and on the University of Pennsylvania School of Engineering and Applied Science Advisory Board. He additionally spent 4 years as an ISAT board member and study lead for the director of DARPA. Mr. Frew currently serves as the chairman of the board for Technology Ventures Corp. (TVC) and formerly served as a director on the board of the ISX Corporation. A lifelong pilot, Mr. Frew holds a commercial pilot's license with ratings in numerous single and multiengine aircraft.

James Larus, director of the eXtreme Computing Group (XCG) in Microsoft Research, has been an active contributor to the programming languages, compiler, and computer architecture communities. He has published many papers and served on numerous program committees and NSF and NRC panels. Dr. Larus became an ACM Fellow in 2006. He joined Microsoft Research as a senior researcher in 1998

to start and, for 5 years, led the Software Productivity Tools (SPT) group, which developed and applied a variety of innovative techniques in static program analysis and constructed tools that found defects (bugs) in software. This group's research has had considerable impact on the research community, as well as being shipped in Microsoft products such as the Static Driver Verifier and FX/Cop and other, widely used internal software development tools. Dr. Larus then became the research area manager for programming languages and tools and started the Singularity research project, which demonstrated that modern programming languages and software engineering techniques could fundamentally improve software architectures. Subsequently, he helped start XCG, which is developing the hardware and software to support cloud computing. Before joining Microsoft, Larus was an assistant and associate professor of computer science at the University of Wisconsin-Madison, where he published approximately 60 research papers and co-led the Wisconsin Wind Tunnel (WWT) research project with Professors Mark Hill and David Wood. WWT was a DARPA- and NSF-funded project that investigated new approaches to simulating, building, and programming parallel shared-memory computers. Larus's research spanned a number of areas, including new and efficient techniques for measuring and recording executing programs' behavior, tools for analyzing and manipulating compiled and linked programs, programming languages for parallel computing, tools for verifying program correctness, and techniques for compiler analysis and optimization. Larus received his M.S. and Ph.D. in computer science from the University of California, Berkeley in 1989, and an A.B. in applied mathematics from Harvard in 1980. At Berkeley, Larus developed one of the first systems to analyze Lisp programs and determine how to best execute them on a parallel computer.

Greg Morrisett is the Allen B. Cutting Professor of Computer Science at Harvard University. His current research interests are in the applications of programming language technology for building secure and reliable systems. In particular, he is interested in applications of advanced type systems, model checkers, certifying compilers, proof-carrying code, and inline reference monitors for building efficient and provably secure systems. He is also interested in the design and application of high-level languages for new or emerging domains, such as sensor networks. Dr. Morrisett received his B.S. degree in mathematics and computer science from the University of Richmond (1989) and his Ph.D. degree in computer science from Carnegie Mellon University (1995). He spent about 7 years on the faculty of the Computer Science Department at Cornell University. In the 2002-2003 academic year, he took a sabbatical at Microsoft's Cambridge Research Laboratory. In January of 2004, he moved to Harvard University.

Walker Royce is vice president and chief software economist at IBM Software Group. Mr. Royce has managed large software engineering projects, consulted with a broad spectrum of IBM's worldwide customer base, and developed software management approaches that exploit an iterative lifecycle, industry best practices, and architecture-first priorities. He is the author of two books: *Software Project Management, A Unified Framework* (Addison Wesley, 1998) and *The Economics of Software Development* (Addison Wesley, 2009). From 1994 through 2009, Mr. Royce was the vice president and general manager of IBM's Worldwide Rational Services organization and led a team of 500 technical specialists in software delivery best practices and $100 million in consulting services. Before joining Rational/IBM, Mr. Royce spent 16 years in software project development, software technology development, and software management roles at TRW Electronics & Defense. He was a recipient of TRW's Chairman's Award for Innovation for his contributions in distributed architecture middleware and iterative software processes in 1990 and was named a TRW Technical Fellow in 1992. He received his B.A. in physics from the University of California and his M.S. in computer information and control engineering from the University of Michigan, and he completed 3 years of further study in computer science at UCLA.

Doug C. Schmidt is the deputy director, Research, and chief technology officer at Carnegie Mellon University's Software Engineering Institute. He was previously a professor at Vanderbilt University, University of California, Irvine, and Washington University St. Louis. He also served as chief technology officer for Zircon Computing and Prism Technologies, where he was responsible for the companies'

technical vision, strategic directions, and growth. In addition, Dr. Schmidt served as a deputy office director and a program manager at DARPA, where he led the national R&D effort on middleware for DRE systems and was the co-chair for the Software Design and Productivity (SDP) Coordinating Group of the U.S. government's multi-agency Information Technology Research and Development (IT R&D) Program, which formulated the multi-agency software research agenda. Dr. Schmidt has published 9 books and over 450 technical papers that cover a range of research topics, including patterns, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded (DRE) middleware and applications running over high-speed networks and embedded system interconnects. In addition to his government service, academic research, and commercial experience, Dr. Schmidt has two decades of experience leading the development of ACE, TAO, CIAO, and CoSMIC, which are widely used, open-source DRE middleware frameworks and model-driven tools that contain a rich set of components and domain-specific languages that implement patterns and product-line architectures for high-performance DRE systems. These technologies have been used successfully by thousands of developers at hundreds of companies worldwide on projects involving medical engineering systems, financial services, datacom/telecom systems, national defense and security systems, and online gaming. Dr. Schmidt has Ph.D. and M.S. degrees in computer science from the University of California, Irvine, and M.A. and B.A degrees in sociology from the College of William and Mary, Williamsburg, VA.

John P. Stenbit is an independent consultant. He recently served as assistant secretary of defense for networks and information integration and as the DoD's chief information officer. Mr. Stenbit has had a career that spans more than 30 years of public- and private-sector service in telecommunications and command and control. In addition to his recent service, his public service includes 2 years as principal deputy director of telecommunications and command and control systems, and 2 years as staff specialist for worldwide command and control systems, both in the Office of the Secretary of Defense. Mr. Stenbit previously was executive vice president at TRW, retiring in May 2001. He joined TRW in 1968 and was responsible for the planning and analysis of advanced satellite surveillance systems. Prior to joining TRW, he held a position with the Aerospace Corporation involving command-and-control systems for missiles and satellites, and satellite data compression and pattern recognition. During this time, he was a Fulbright Fellow and Aerospace Corporation Fellow at the Technische Hogeschool, Einhoven, the Netherlands, concentrating on coding theory and data compression. He has served on numerous scientific boards and advisory committees, including as chair of the Science and Technology Advisory Panel to the Director of Central Intelligence and as a member of the Science Advisory Group to the Directors of Naval Intelligence and the Defense Communications Agency. He is a member of the National Academy of Engineering.

Kevin J. Sullivan is an associate professor and a Virginia Engineering Foundation (VEF) Endowed Faculty Fellow in computer science at the University of Virginia, where he has worked since 1994. His research interests are mainly in software engineering and languages. He has served as associate editor for the *Journal of Empirical Software Engineering* and the *ACM Transactions on Software Engineering and Methodology* and on the program and executive committees of conferences including the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), the International Conference on Software Engineering (ICSE), Aspect-Oriented Software Development (AOSD), and the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). He and his students are broadly interested in the design and engineering of software-intensive systems, with an emphasis on the need for a value-based theory and practice of system design. Dr. Sullivan received his undergraduate degree from Tufts University in 1987 and M.S. and Ph.D. degrees in computer science and engineering from the University of Washington in 1994.