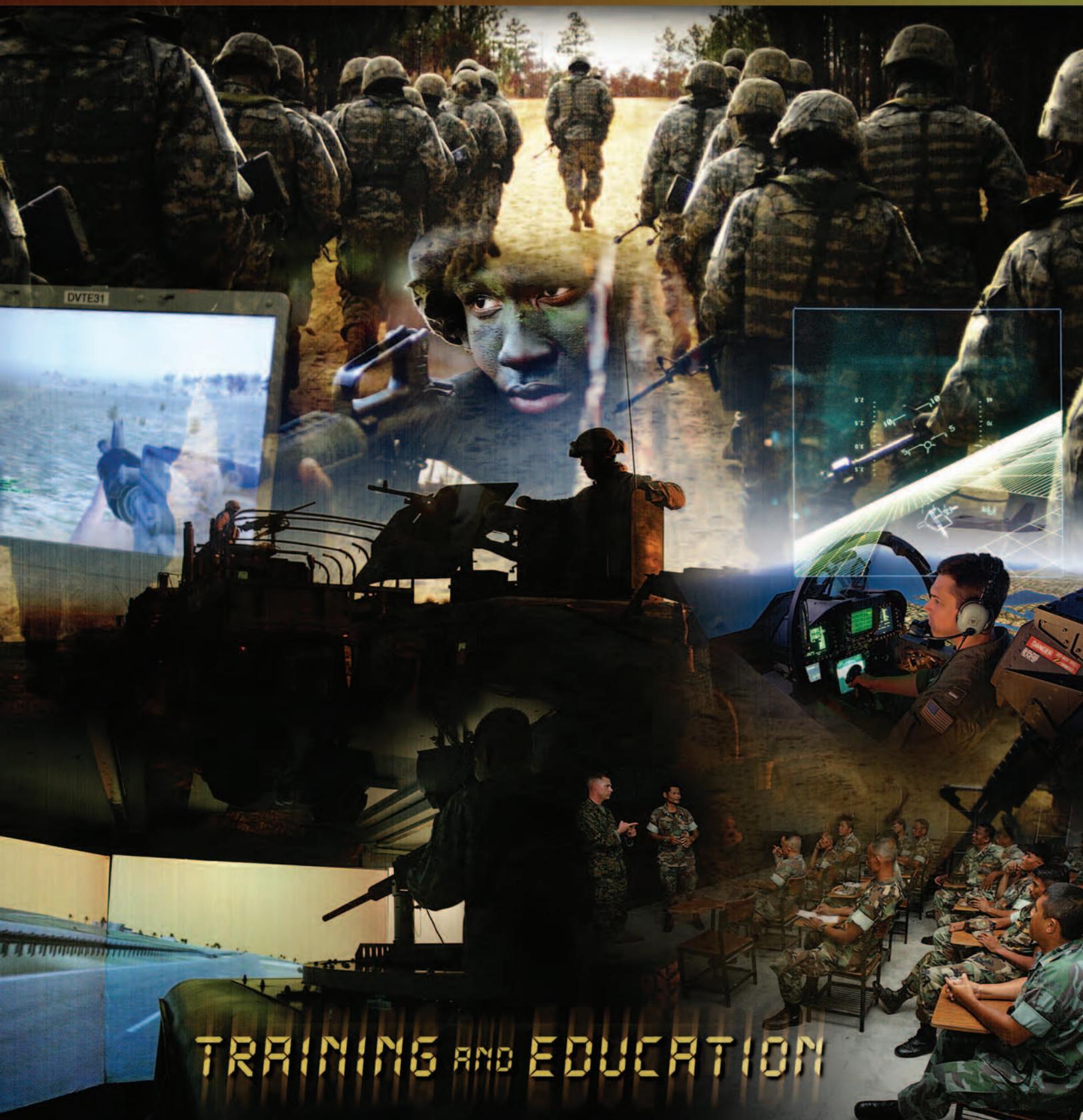


CROSSTALK

January 2008 *The Journal of Defense Software Engineering* Vol. 21 No. 1



TRAINING AND EDUCATION

Report Documentation Page

*Form Approved
OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE JAN 2008		2. REPORT TYPE		3. DATES COVERED 00-00-2008 to 00-00-2008	
4. TITLE AND SUBTITLE CrossTalk: The Journal of Defense Software Engineering. Volume 21, Number 1, January 2008				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

Policies, News, and Updates

4 Announcing CROSSTALK's Co-Sponsor Team for 2008

Meet CROSSTALK's 2008 co-sponsors.

5 The 2008 CROSSTALK Editorial Board

Here is a list of CROSSTALK's article reviewers.

Training and Education

6 The Critical Need for Software Engineering Education

Long describes the need for more dedicated software engineering educational programs and professional software engineering certification programs in the United States.

by Dr. Lyle N. Long

11 Using Inspections to Teach Requirements Validation

This article describes an experiment conducted in a graduate-level requirements engineering course to provide students a real world experience in requirements validation.

by Lulu He, Dr. Jeffrey C. Carver, and Dr. Rayford B. Vaughn

16 Integrating Software Assurance Knowledge Into Conventional Curricula

This article discusses the results of a comparison of the Common Body of Knowledge for Secure Software Assurance with traditional computing disciplines.

by Dr. Nancy R. Mead, Dr. Dan Shoemaker, and Jeffrey A. Ingalsbe

21 Software Engineering Continuing Education at a Price You Can Afford

Maj. Bohn gives detailed information about the Air Force Institute of Technology's Software Professional Development Program.

by Maj Christopher Bohn, Ph.D.

Software Engineering Technology

23 How to Avoid Software Inspection Failure and Achieve Ongoing Benefits

This article describes the proven benefits of inspections and signifies that they are too significant to let them fall by the wayside.

by Roger Stewart and Lew Priven

Open Forum

28 Computer Science Education: Where Are the Software Engineers of Tomorrow?

This article briefly examines the set of programming skills that should be part of every software professional's repertoire.

by Dr. Robert B.K. Dewar and Dr. Edmond Schonberg



Cover Design by
Kent Bingham

Additional art services
provided by Janna Jensen

Departments

3 From the Sponsor

10 Coming Events
Letter to the Editor

20 SSTC 2008 Ad

31 BACKTALK

CROSSTALK

Co-SPONSORS:

DoD-CIO *The Honorable John Grimes*

OSD (AT&L) *Kristen Baldwin*

NAVAIR *Jeff Schwalb*

76 SMXG *Kevin Stamey*

309 SMXG *Norman LeClair*

DHS *Joe Jarzombek*

STAFF:

MANAGING DIRECTOR *Brent Baxter*

PUBLISHER *Elizabeth Starrett*

MANAGING EDITOR *Kase Johnston*

ASSOCIATE EDITOR *Chelene Fortier-Lozancich*

ARTICLE COORDINATOR *Nicole Kentta*

PHONE (801) 775-5555

E-MAIL crosstalk.staff@hill.af.mil

CROSSTALK ONLINE www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); and Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.



Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 30.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See www.stsc.hill.af.mil/crosstalk, call (801) 777-0857 or e-mail stsc.web.master@hill.af.mil.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.



Training and Education



One of my favorite resources is Webster's 1828 Dictionary. Unlike any of our modern dictionaries, the 1828 version provides a deeper understanding of words. I thought the 1828 definition of education was quite fitting for this month's theme. Webster's says that education comprehends all that series of instruction and discipline which is intended to enlighten understanding ... and fit them for usefulness in their future stations.

A primary question this issue attempts to highlight is *do our educational institutions in fact prepare software professionals to be successful in their future stations?* In October 2006, a study conducted by the Center for Strategic and International Studies revealed a significant shortfall in the number of software professionals that had been formally educated in software project management. The study indicated that our industry is lacking in program managers, software architects, systems engineers, and domain experts. Many industry experts agree that there is an adequate supply of programmers, but the pool of these critical senior managers and system experts is very limited.

As Department of Defense (DoD) systems become more and more software intensive, software developments get bigger, more complicated, and more dependent on senior software professionals to get the project on the right path and keep it there. Since studies seem to indicate that we are falling behind in our attempt to educate certain pockets of critical expertise and our software projects are in greater need of these same professionals, our problem is getting worse. I hope this month's articles will inspire you to think about how you will address this trend in your organization.

For a more thorough discussion of this trend from the perspective of aerospace engineering, I hope you will read *The Critical Need for Software Engineering Education* by Dr. Lyle N. Long. You will find more specific software educational issues addressed in the two articles that follow. In *Using Inspections to Teach Requirements Validation* by Lulu He, Dr. Jeffrey C. Carver, and Dr. Rayford B. Vaughn, the authors share an experiment that compares the use of both checklists and Perspective-Based Reading to aid in a requirements validation exercise. In *Integrating Software Assurance Knowledge Into Conventional Curricula* by Dr. Nancy R. Mead, Dr. Dan Shoemaker, and Jeffrey A. Ingalsbe, the authors compare the contents of the Common Body of Knowledge for Secure Software Assurance with the Computing Curricula 2005: The Overview Report. Their intent is to map our security needs with software-related curricula being recommended for our schools.

One of the options available to assist DoD readers with required software training is the software curriculums available through the Air Force Institute of Technology (AFIT); you can read more about AFIT in *Software Engineering Continuing Education at a Price You Can Afford* by Maj Christopher Bohn, Ph.D. You will find another thought-provoking article from Roger Stewart and Lew Priven with their discussion of successfully implementing software inspections in *How to Avoid Software Inspection Failure and Achieve Ongoing Quality, Cost, and Schedule Benefits*. We conclude this month's issue with an insightful article by Dr. Robert B.K. Dewar and Dr. Edmond Schonberg entitled *Computer Science Education: Where Are the Software Engineers of Tomorrow?*

It should be clear throughout the software community that education does not end once a graduate has a bachelor's degree. The technology we work with continues to grow at an astounding rate. Even if additional degrees are not sought, all professionals need additional education to keep current with the needs of the software community. As a co-sponsor for CROSSTALK, I am happy to provide one additional source for this education.

Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor



Announcing CROSSTALK's Co-Sponsor Team for 2008

Elizabeth Starrett
CROSSTALK

I wish to express my continuing gratitude for the support CROSSTALK's co-sponsors again provide in 2008. I know our readers and staff appreciate the benefits provided to the software community by the information made available as a result of their sponsorship. Co-sponsor team members are identified below with a description of their organization. Please look for their contributions each month in our From the Sponsor column found on page 3. Their organizations will also be highlighted on the back cover of each issue of CROSSTALK.



The Honorable John Grimes, Department of Defense – Chief Information Officer

The Assistant Secretary of Defense for Networks and Information Integration for the Department of Defense Chief Information Officer (ASD[NII]/DoD-CIO) is the principal staff assistant and advisor to the Secretary on networks and net-centric policies and concepts, command and control, communications, non-intelligence space matters, enterprise-wide integration of DoD information matters, and Information Technology. Additionally, the DoD-CIO has responsibilities for integrating information and related activities and services across the DoD. The mission of the organization is to enable Net-Centric operations. NII/CIO is leading the Information Age transformation that will enhance the DoD's efficiency and effectiveness by establishing an Information on Demand capability. See <www.dod.mil/cio-nii> for more information.

that meet Fleet needs, providing a technological edge over adversaries. See <www.navair.navy.mil> for more information.



Kevin Stamey, 76 SMXG Director

The 76th Software Maintenance Group at the Oklahoma City-Air Logistics Center is a leader in the avionics software industry that understands total system integration. The center has a proven record of producing software on time, on budget, and defect-free. Its people provide the expertise, software, weapons, interface, and aircraft systems that are fully integrated to ensure dependable war-winning capabilities. Its areas of expertise include navigation, radar, weapons and system integration, systems engineering, operational flight software, automatic test equipment, and more. For more information, see <www.bringittotinker.com>.



Kristen Baldwin, Office of the Secretary of Defense – Acquisition, Technology and Logistics

The Office of the Secretary of Defense for Acquisition, Technology and Logistics (OSD(AT&L)) Software Engineering and Systems Assurance is the staff agent responsible for all matters relating to DoD software engineering, systems assurance, system of systems (SoS) engineering, and Capability Maturity Model Integration. Organizational focus areas include policy, guidance, education and training, acquisition program support, software acquisition management and development techniques, software and systems engineering integration, SoS enablers and best practices, engineering for system assurance, and government-industry collaboration. See <www.acq.osd.mil/sse/ssa> for more information.



Norman LeClair, 309 SMXG Acting Director

The 309th Software Maintenance Group at the Ogden-Air Logistics Center is a recognized world leader in cradle-to-grave systems support, encompassing hardware engineering, software engineering, systems engineering, data management, consulting, and much more. The division is a Software Engineering Institute Software Capability Maturity Model® (CMM®) Integration Level 5 organization with Team Software ProcessSM engineers. Their accreditations also include AS 9100 and ISO 9000. See <www.mas.hill.af.mil> for more information.



Terry Clark, NAVAIR, Systems Engineering Department – Director, Software Engineering

The Naval Air Systems Command (NAVAIR) has three Strategic Priorities through which it produces results for the Sailor and the Marine. First are its *People* that we develop and provide tools, infrastructure, and processes needed to do their work effectively. Next, *Current Readiness* that delivers NAVAL aviation units ready for tasking with the right capability, at the right time, and the right cost. Finally, *Future Capability* in the delivery of new aircraft, weapons, and systems on time and within budget,



Joe Jarzombek, Department of Homeland Security – Director of Software Assurance

The DHS National Cyber Security Division serves as a focal point for software assurance (SwA), facilitating national public-private efforts to promulgate best practices and methodologies that promote integrity, security, and reliability in software development and acquisition. Collaborative efforts of the SwA community have produced several publicly available online resources. For more information, see the Build Security In Web site <<https://buildsecurityin.us-cert.gov>>, which is expanding to become the SwA Community of Practice portal <www.us-cert.gov/swa> to provide coverage of topics relevant to the broader stakeholder community.

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SM Team Software Process are service marks of Carnegie Mellon University.

For more information about becoming a CROSSTALK co-sponsor, please contact Elizabeth Starrett at (801) 775-4158 or <beth.starrett@hill.af.mil>.

The 2008 CROSSTALK Editorial Board

Elizabeth Starrett
CROSSTALK

CROSSTALK proudly presents the 2008 CROSSTALK Editorial Board. Each article submitted to CROSSTALK is reviewed by two technical reviewers from the list below in addition to me, CROSSTALK's publisher. The insights from the board improve the readability and usefulness of the articles that are published in CROSSTALK. Most reviewers listed have graciously offered their own time to support CROSSTALK's technical review process. We give very special thanks to all those participating in our 2008 CROSSTALK Editorial Board.

COL Ken Alford, Ph.D.	National Defense University
Bruce Allgood	Electronic Systems Center Engineering Directorate
Brent Baxter	Software Technology Support Center
Jim Belford	Software Technology Support Center
Dr. Alistair Cockburn	Humans and Technology
Richard Conn	Retired (formerly with Microsoft)
Dr. David A. Cook	The AEgis Technologies Group, Inc.
Les Dupaix	Software Technology Support Center
Sally Edwards	Information Technology Standards and Solutions
Robert W. Ferguson	Software Engineering Institute
Dr. John A. "Drew" Hamilton Jr.	Auburn University
Tony Henderson	Software Technology Support Center
Lt. Col. Brian Hermann, Ph.D.	Olin Institute for Strategic Studies, Harvard University
Thayne Hill	Software Technology Support Center
George Jackelen, PMP	GAITS, Inc.
Deb Jacobs	Priority Technologies, Inc.
Dr. Randall Jensen	Software Technology Support Center
Alan C. Jost	Raytheon
Daniel Keth	Software Technology Support Center
Paul Kimmerly	U.S. Marine Corps
Theron Leishman	Northrop Grumman
Glen L. Luke	519th Combat Sustainment Squadron
Gabriel Mata	Software Technology Support Center
Paul McMahan	PEM Systems
Dr. Max H. Miller	Raytheon Integrated Defense Systems
Mike Olsem	Science Applications International Corporation
Glenn Palmer	L-3 Communications, Inc.
Doug J. Parsons	Army PEO Simulation, Training and Instrumentation
Tim Perkins	Battle Control System-Fixed Sustainment Office
Gary A. Petersen	Arrowpoint Solutions, Inc.
Vern Phipps	SabiOso
David Putman	309th Software Maintenance Group
Kevin Richins	Shim Enterprise, Inc.
Thom Rodgers	Software Technology Support Center
Larry Smith	Software Technology Support Center
Elizabeth Starrett	Software Technology Support Center
Tracy Stauder	309th Software Maintenance Group
COL John "Buck" Surdu, Ph.D.	Defense Advanced Research Projects Agency
Kasey Thompson	Software Technology Support Center
Dr. Will Tracz	Lockheed Martin Integrated Systems and Solutions
Jim Van Buren	Charles Stark Draper Laboratory
Dr. Rayford B. Vaughn Jr.	Mississippi State University
David R. Webb	309th Software Maintenance Group
Mark Woolsey	Software Technology Support Center
David Zubrow	Software Engineering Institute



The Critical Need for Software Engineering Education

Dr. Lyle N. Long

The Pennsylvania State University

Software affects almost every aspect of our daily lives (manufacturing, banking, travel, communications, defense, medicine, research, government, education, entertainment, law, etc.). It is an essential part of our military systems, and it is used throughout the civilian sector, including safety-critical and mission-critical systems. In addition, the complexity of many of these systems has been growing exponentially. Unfortunately, the U.S. higher education system has not kept pace with these needs. Existing undergraduate and graduate science and engineering programs need to incorporate more material on software engineering. This is especially true for aerospace engineering, since those systems rely heavily on computation, information, communications, and software. In addition, the United States needs more dedicated software engineering educational programs and professional software engineering certification programs.

Software is everywhere, from cell phones to large military systems. According to the National Academy of Science [1], "... software is not merely an essential market commodity but, in fact, embodies the economy's production function itself." The National Institute of Standards and Technology (NIST) estimates that software errors cost the U.S. economy \$59.5 billion a year and software sales accounted for \$180 billion [2]. Software engineering education does not get the attention it deserves, even though it is crucially important to our economy. The issues related to software engineering as a discipline and the debates which have occurred over the years, are not new and are described in [3, 4, 5].

The list of software disasters grows each year. Some of the best-known include the following: the Ariane 5 rocket (Flight 501) [6, 7], the Federal Bureau of Investigation Virtual Case File system [8], the Federal Aviation Administration Advanced Automation System [7, 9], the California Department of Motor Vehicle system, the American Airlines reservation system, and many, many more [7, 10]. The F-22 aircraft also had problems initially due to its complex software systems. Software disasters cost the United States billions of dollars every year, and this may only get worse since future systems will be more complex. Boeing spent roughly \$800 million on software for the 777, and they might need to spend five times that on the 787 [11]. Aerospace systems will also include some levels of autonomy, accompanied by an entirely new level of software complexity. To help prevent future disasters, we must have more software engineers trained in rigorous technical programs that are on par with other engineering programs. The United States

should not wait until there is a disaster that causes large numbers of human casualties before it acts. We do not currently have enough software engineers. We need to educate many more in the near future, especially considering the large group of engineers that will be retiring in the next 10 years [12]. In 2005, the average age of an aerospace engineer was 54 [13]. In addition, more than 26 percent of the aerospace workers will be eligible for retirement in 2008 [14].

Importance of Software in Aerospace Systems

The aerospace industry provides roughly \$900 billion in economic activity and accounts for more than 15 percent of the gross domestic product and supports more than 15 million high quality jobs in the United States [14]. These aerospace systems rely heavily on software, which has been called the Achilles Heel of aerospace systems. There are numerous anecdotes and examples that illustrate the importance of computing and software in aerospace. For example:

- The Boeing 777 has 1,280 onboard processors that use more than four million lines of software; Ada accounts for 99.5 percent of this [15, 16].
- The F-22 has more than two million lines of software onboard; between 80 and 85 percent is in Ada [17].
- Some Blackhawk helicopters have almost 2,000 pounds of wire connecting all the computers and sensors.
- The wiring harness is often more complex and more difficult to design than the aircraft structure.
- Some aircraft cannot fly without their onboard computers (e.g., F-16 and F-117).
- The air traffic control system relies

heavily on computers, software, and communications.

- Interplanetary robotics and spacecraft perform amazing feats, often in extreme environments.
- Autonomous, intelligent, unmanned vehicles will be even less deterministic than current systems.
- Computers are also important in the design and analysis of aerospace systems. Often this means using high-performance, massively parallel computer systems.
- Communication systems are critically important for aircraft and spacecraft; this now includes computer networking onboard, to the ground, and to other aerospace vehicles.
- Modern aircraft and spacecraft seldom work alone – they are usually part of a system of systems.

One way to measure the need for software engineers in the aerospace field is to research existing job opportunities. An Oct. 2006 review of the Lockheed-Martin Corporation Web site showed they had 536 job openings for recent graduates, including 68 openings (13 percent) in software engineering and four in aerospace engineering. Most of the aerospace engineering job openings were for structural engineers capable of performing finite element analyses. It should be noted that they do hire aerospace engineers in other areas as well (for example, aerospace control experts are sometimes listed under embedded systems). The Boeing employment Web page gave similar results. They had 298 job openings related to software. There were only three jobs that mentioned *aerodynamics* (none of which were actually jobs for aerodynamicists). When searching the Boeing site for *aerospace engineer*, it returned a listing of six open positions in

structural engineering. This is probably an indication that aerospace engineering educational programs are concentrating too much on the applied physics of aerospace engineering and not enough on computing and software. We need to work with industry and the government to redefine aerospace engineering. We need to educate students capable of designing and building the new aerospace systems that we will need in the future – which will be dominated by computing, networking, and information systems.

Software Engineering Defined

The Institute of Electrical and Electronics Engineers (IEEE) defines software engineering [3] as “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.” A good summary of software engineering can be found in [18].

Software systems are some of the most complicated things humans have ever created. To design and build them, one needs to follow processes and procedures typical of other engineering disciplines [6, 19]. First, the requirements need to be carefully defined. Then the architecture of the software system needs to be developed. Once the requirements and architecture are defined, one can begin code development. The code then needs verification, validation, and testing. There are many ways of accomplishing all of these steps which are related to the type of life-cycle model used and the type of system developed. In addition, one needs to consider how to estimate costs, how to manage the people, and how to monitor the ethical responsibilities of the team. This is not unlike the steps required to design and build any complex system (e.g., bridges, aircraft, and computer hardware). The actual code development or programming can be a fairly small portion of the process [6].

Most engineers and scientists do not fully appreciate or understand software engineering. Even high school students think they can do software after they learn the basics of Java or C++ syntax. All too often, software engineering is equated with programming. This is like equating civil engineering with pouring concrete. Many people can pour concrete, but few are civil engineers and can build large, technically inspired masterpieces. Likewise, many people can program, but few can develop large software masterpieces. It is not uncommon to hear people arguing about the merits or drawbacks of the different computer languages, even though they are not well versed on the var-

ious languages. Often, they simply like the language that they grew up with and do not appreciate or understand the others. These misconceptions are especially apparent in discussions regarding Ada [20], which is still probably the best language to use for mission- or safety-critical systems. In reality, people who develop code without sound software engineering approaches are merely hackers. Of course, programmers are an essential part of software engineering, and talented programmers are quite rare and extremely valuable.

Software Engineering Education

Both the U.S. economy and national defense depend upon software, but many of these large software systems are being developed by people who have never been formally trained in software engineering. While there are some incredibly talented self-taught software engineers, we should

“We need to work with industry and the government to redefine aerospace engineering. We need to educate students capable of designing and building the new aerospace systems that we will need in the future ...”

not rely on the majority of our software engineers being self-taught. We would never build modern aircraft without aerospace engineers, and we would never build bridges or buildings without civil engineers. So why are we developing large software systems without teams of formally trained and professionally certified software engineers?

Recently, Dr. John Knight, a professor at the University of Virginia, contrasted software engineering to other engineering disciplines [21]. He spoke of how 1,000-year-old cathedrals were built using the best civil engineering technology of the time and how these buildings are still standing. Civil engineering has evolved tremendously over the ages, and now we have enormous skyscrapers and spectacu-

lar bridges. This would not be possible without a vibrant civil engineering educational system, research programs, and mentoring. Similar analogies could be drawn from other engineering disciplines. A thousand years from now, people will be marveling at aerospace engineering milestones such as the Wright Flyer, the SR-71 Blackbird, and the Apollo program. All were great engineering projects in their day and are now in museums. Will there be any software cathedrals to marvel at 1,000 years from now? Or will future generations view us as hackers?

Traditional Science and Engineering Educational Programs

Many students who graduate from U.S. science and engineering programs will eventually work in software development. Unfortunately, most of them will get little or no software education. For example, 24 percent of physics graduates will be working on software five to eight years after graduation [22]. Most of them will probably receive no training in software engineering in college. Other science graduates, even outside of engineering, may also eventually work in software development. It would be very beneficial for these students to know more about software engineering before they graduate. They need more than a freshman-level course in programming. This is true of almost all the traditional science and engineering degree programs.

It is also not valid to assume that computer science graduates are software engineers, either. It is fairly easy to graduate from a computer science program with very little education in software development. Knight and Leveson describe the need for more software education in computer science and computer engineering programs and advocate for more software engineering programs [23].

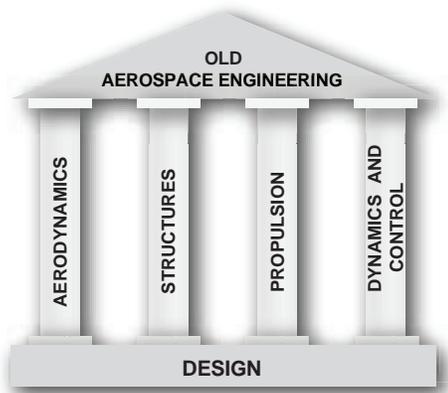
The need for software education is especially critical in aerospace engineering programs. Aerospace engineers have always prided themselves on being the system integrators, but to do this you must have some understanding of the complete aerospace system you are developing. In modern combat aircraft, the electronic components account for roughly 10 percent of the weight and 33 percent of the cost [24]. So if aerospace engineers are not well versed in computing, networking, sensors, and software then they cannot understand the complete system (unless that system is 60 years old). Students need to be trained so that they can develop the next generation of aerospace systems, not old aircraft and old

spacecraft. Aerospace systems have always used the latest technology to achieve amazing performance. Future and current systems rely heavily on computers and software and students need to know that. Aerospace engineers are needed as system integrators, but this is only possible if they have some understanding of the complete system (including computing and software).

Today this goes beyond the onboard avionics since modern aircraft and spacecraft are almost always tied to other systems, but avionics is a huge part of aerospace systems. Processing power and computer memory have been increasing exponentially in military aircraft since about 1960 [25]. The F-106 had less than 20 kilobytes of memory, while the Joint Strike Fighter (JSF) could have more than two gigabytes. Avionics could account for 40 percent of the cost of the JSF. The report also states that software content in these systems has increased dramatically, and that we need more software engineers.

Computing and software are integral parts of aerospace engineering. It is now one of the key disciplines in aerospace engineering. Traditionally, aerospace engineering [26] was built upon four technology *pillars*: aerodynamics, structures, propulsion, and dynamics and control, as shown in Figure 1. These pillars are reflected in aerospace engineering curricula. All these disciplines were important for the Wright brothers and for every aerospace system since then. However, modern aerospace engineering must include five pillars, as shown in Figure 2. In [27], the authors refer to the five areas as the following: aerodynamics, materials, avionics, propulsion, and controls. Current and future aerospace systems are and will be designed using computers. They will have onboard computers and will need to communicate with other vehicles and computers.

Figure 1: *Old Aerospace Engineering*

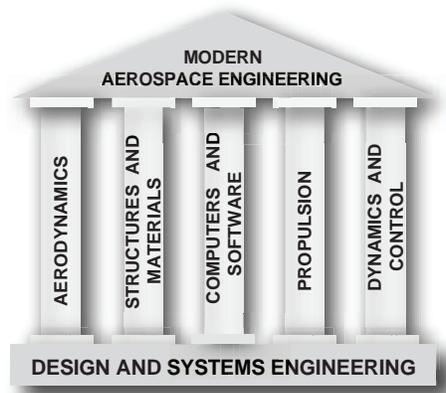


Computing (including processing, networking, and storing data) and software are essential elements of aerospace engineering, and they are the fifth pillar. In addition, this fifth pillar might be the most important pillar, and it is far less mature than the other four. Aerospace engineering educational programs have a strong emphasis on applied physics (e.g., fluid dynamics, structural dynamics, dynamics, combustion, and propulsion). Historically, there were good reasons for this, but we cannot continue to neglect the research and educational needs in aerospace computing and software.

“While computing and software is crucial for aerospace systems, existing aerospace engineering educational programs usually do not reflect this fact.”

While computing and software is crucial for aerospace systems, existing aerospace engineering educational programs usually do not reflect this fact. Most aerospace engineering programs require roughly 40 courses over a four-year period, but students often take only one course related to software (a freshman-level programming course). Also, there is usually no requirement to learn about avionics, embedded systems, networking, sensors, or computer hardware. This trend carries through to aerospace engineering graduate programs as well, where the entrance exams and curricula seldom include computing, software, or avionics. They are often primarily

Figure 2: *Modern Aerospace Engineering*



applied physics programs.

Pennsylvania State University has been working to modernize its aerospace engineering curricula [28]. The university now offers senior-level courses in advanced computer programming (object-oriented programming, Java, C++, Ada, etc.) and software engineering (using [6]), both for aerospace engineers. Penn State also has a new course on the Global Positioning System. As of 2006, aerospace engineering students will be *required* to take either the software engineering course or an electronic circuits course. Ideally, they should take both and also be exposed to systems engineering, embedded systems, networking, information systems, sensors, and software. These additional topics could be covered in their technical electives or in graduate courses. Some of them could be covered in a minor also. The university also hopes to establish an undergraduate minor in information sciences and technology for aerospace engineering in 2008.

It should also be noted that it is difficult to teach an engineer all they need to know in four years. In fact, the U.S. National Academy of Engineering [29] recommends that the bachelor of science degree be recognized as a *pre-engineering degree*. Scientists and engineers need to continue learning throughout their lifetimes to be effective. In addition, many aerospace engineering positions require a master's degree, which allows the student to concentrate on a particular area. An excellent combination would be for a student to get a bachelor of science in aerospace engineering and then a master's degree or doctorate in software (or systems) engineering. These graduates would be extremely valuable. Another possibility is to offer an undergraduate or graduate minor in software engineering. Penn State has a popular graduate minor in computational science, which attracts students from a wide variety of science and engineering departments [30]. A similar program could be created for software engineering or systems engineering.

Dedicated Software Engineering Education Degree Programs

As previously stated, existing science and engineering education programs need to include more computing and software in their curricula, but they also need more dedicated software engineering programs. These software engineering programs, however, need to include

plenty of science and engineering in their curricula (e.g., physics, mathematics, and embedded systems). They should not have an overemphasis on management, business, and processes. The Association for Computing Machinery (ACM), the IEEE, and the National Science Foundation have developed very good undergraduate curricula in software engineering [3].

Currently in the United States, there are only 10 accredited software engineering undergraduate programs [31], while there are 67 aerospace engineering programs. The United States needs many more software engineering programs. This needs to happen soon, since it takes years to start new programs and for students to graduate. In addition, the United States has an aging workforce. Some companies in the aerospace and defense business could see 40 percent of their workforce retire in the next five years [12]. According to the Wall Street Journal, organizations such as the National Aeronautics and Space Administration have more engineers over 60 than under 30.

In addition to the existing undergraduate software engineering programs, there are 109 software engineering master's degree programs and 40 software engineering doctorate programs in the United States. Few of the undergraduate or graduate programs, however, are at major research universities. In addition, few of them exist at universities included in the top 25 schools listed in the *U.S. News and World Report* rankings. Most of these programs are at relatively small schools, maybe because they are able to react more quickly to industry and society needs.

Unfortunately, change occurs extremely slowly in academia because there are few incentives to change. Government funding could and should be used to help expedite these changes. Industry leaders need to get involved and demand change as well. There needs to be internships and mentoring available. There is also a need for continuing education. At the government labs and in industry, there is a huge need for software engineering training for its existing workforce.

We also need software engineering professional certification. The IEEE has developed an excellent Certified Software Development Professional (CSDP) program [32, 33]. This is a great program, but it is not quite a software engineering certification program. Unfortunately, there is no requirement

for a science or engineering background for the certification, so it is not the same as other professional engineering certification programs. In addition, at the time this article was written, there were only 575 people in the world who have the CSDP certification. Beginning in 1999, Texas began certifying software engineers [34]. In addition, the Open Group has established an information technology architect certification program [35].

Conclusion

Higher education in the United States needs to be more responsive to the software engineering needs of its industry and government labs. The United States cannot be complacent with its technological leads in any field, especially software and aerospace, which are two of the most important industries in its economy. These two industries annually provide more than \$180 billion and \$900 billion, respectively, to the economy. Technology has been changing at an exponential rate, and too often curricula changes extremely slowly. Software engineering needs to be incorporated into existing science and engineering programs, especially aerospace engineering curricula. We also need to create more dedicated software engineering educational programs at all levels – short courses, bachelors, masters, and doctorate levels. And finally, there also needs to be a national effort to develop professional certification of software engineers. ♦

References

1. Jorgenson, D.W., and C.W. Wessner, Eds. Measuring and Sustaining the New Economy, Software, Growth, and the Future of the U.S. Economy. Washington, D.C.: National Academies Press, 2006.
2. NIST <www.nist.gov/public_affairs/releases/n02-10.htm>.
3. IEEE Computer Society and the ACM. "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering." 2004 <<http://sites.computer.org/ccse/SE2004Volume.pdf>>.
4. Vaughn, R. "Software Engineering Degree Programs." CROSSTALK Mar. 2000.
5. Computer Science and Telecommunications Board. Expanding Information Technology Research to Meet Society's Needs. Washington, D.C.: The National Academies Press, 2000.
6. Sommerville, I. Software Engineering. Addison-Wesley, 2006.
7. Glass, Robert L. Software Runaways:

Monumental Software Disasters. Prentice-Hall, 1997.

8. Eggen, D., and G. Witte. "The FBI's Upgrade That Wasn't." Washington Post 18 Aug. 2006.
9. U.S. House of Representatives. Proc. of the Aviation Subcommittee Meeting. Washington, D.C.: 2001.
10. Leveson, Nancy G. "Role of Software in Spacecraft Accidents." Journal of Spacecraft and Rockets 41.4 (2004).
11. Winter, D.C. Presentation at the National Workshop on Aviation Software Systems: Design for Certifiably Dependable Systems. Alexandria, VA: Oct. 5-6, 2006.
12. "The Aging Workforce: Turning Boomers Into Boomerangs." The Economist 16 Feb. 2006.
13. Albaugh, J.F. "Embracing Risk: A Vision for Aerospace in the 21st Century." Frank Whittle Lecture, Royal Aeronautical Society, Jan. 19, 2005.
14. Commission on the Future of the U.S. Aerospace Industry. "Final Report of the Commission on the Future of the U.S. Aerospace Industry." Washington D.C.: National Academies Press, 2002.
15. Hafner, K. "Honey, I Programmed the Blanket." New York Times 27 May 1999.
16. Pehrson, R.J. "Software Development for the Boeing 777." CROSSTALK Jan. 1996.
17. Moody, B.L. "F-22 Software Risk Reduction." CROSSTALK May, 2000.
18. U.S. Air Force. Software Technology Support Center (STSC). "A Gentle Introduction to Software Engineering." Hill Air Force Base, UT: STSC, 1999.
19. Glass, R.L. Facts and Fallacies of Software Engineering. Addison-Wesley, 2006.
20. Ada Home <www.adahome.com>.
21. Knight, J.C. Presentation at the National Workshop on Aviation Software Systems: Design for Certifiably Dependable Systems. Alexandria, VA: Oct. 5-6, 2006.
22. American Institute of Physics. The Statistical Research Center <www.aip.org/statistics>.
23. Knight, J.C., and N.G. Leveson. "Software and Higher Education Inside Risks Column." Communications of the ACM 49.1 (2006).
24. Sanders, P. "Improving Software Engineering Practice." CROSSTALK Jan. 1999.
25. Aging Avionics in Military Aircraft. Washington D.C.: National Academies Press, 1993.
26. Long, L.N. "Computing, Information,

COMING EVENTS

February 4-8

28th Annual Texas Computer Education Association's Convention and Exposition
Austin, TX
www.tcea2008.org

February 13-15

2008 Conference for Industry and Education Collaboration
New Orleans, LA
www.asee.org/conferences/ciec/2008/index.cfm

February 18-22

2008 Working IEEE/IFIP Conference on Software Architecture
Vancouver, BC, Canada
www.wicsa.net

February 25-28

24th Annual National Test and Evaluation Conference
Palm Springs, CA
www.ndia.org/Template.cfm?Section=8910&Template=/ContentManagement/ContentDisplay.cfm&ContentID=18912

February 27-28

AFCEA 2008 Homeland Security Conference
Washington, D.C.
www.afcea.org/events/homeland/landing.asp

April 29-May 2



2008 Systems and Software Technology Conference
Las Vegas, NV
www.sstc-online.org

COMING EVENTS: Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.

and Communication: The Fifth Pillar of Aerospace Engineering." Journal of Aerospace Computing, Information, and Communication 1.1 (2004).

27. "The Future of Aerospace." Washington, D.C.: National Academies Press, 1993.
28. Pennsylvania State University Curriculum Guide 2006-2007 <www.aero.psu.edu/undergrads/UG_Curriculum_Guide_2006-07.pdf>.
29. "Educating the Engineer of 2020: Adapting Engineering Education to the New Century." Washington D.C.: National Academies Press, 2005.
30. <www.csci.psu.edu/minor.html>.
31. ABET <www.abet.org/>.
32. IEEE Computer Society. IEEE Computer Society Certified Software Development Professional Program Web Center <www.computer.org/portal/pages/ieeecs/education/certification/>.
33. IEEE Computer Society. Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, 2004.
34. Voas, J. "The Software Quality Certification Triangle." CROSSTALK Nov., 1998.
35. The Open Group. "IT Architect Certification Program" <www.open.group.org/itac>.

About the Author



Lyle N. Long, D.Sc., is a distinguished professor of aerospace engineering, bioengineering, and mathematics at the Pennsylvania State University.

He is also director of the graduate minor in computational science. Long has a doctorate of science from George Washington University, a master's degree from Stanford University, and a bachelor's degree in mechanical engineering from the University of Minnesota. He is an IEEE Computer Society Certified Software Development Professional. Long received the 1993 IEEE Computer Society Gordon Bell Prize. He is a Fellow of the American Institute of Aeronautics and Astronautics, is a senior member of the IEEE, and has written more than 130 technical papers. In 2007-2008, he was a Moore Distinguished Scholar at the California Institute of Technology.

Pennsylvania State University
233 Hammond BLDG
University Park, PA 16802
Phone: (814) 865-1172
Fax: (814) 865-7092
E-mail: lnl@psu.edu

LETTER TO THE EDITOR

Dear CROSSTALK Editor,

Once again, the cover for CROSSTALK is a *knockout!* Congratulations to the staff and to the editor for a great edition. I was wondering when systems engineering would be a *transitional* topic about systems engineering and software inclusion.

When I stepped down from the principal investigator position on the B-1B program, I went into software because of the lack of understanding I found between systems engineering and software. Due to my relationship with the company's system engineering manager and the software manager – and the help of CROSSTALK articles that I sent them – they finally decided to have meetings in their manager's office and talked together! What a milestone *that was!* I still send CROSSTALK articles to these guys, which they are grateful for because it helps keep them current on industry

thinking.

This edition should be given to every systems engineer manager and staff, as well as software managers. Let's get the communication going among our contractors!

I could not have been more pleased with Dr. John W. Fischer's introduction in the Sponsor's Note to this month's issue of CROSSTALK. His remarks are dead-on regarding the issues and *evolution* of system engineering with software. Gee, can you imagine what the foundation for requirements would start to look like?

Thanks for another great issue!

– **Melonee Moses**
Software/Logistics Management Specialist
DCMA Boeing
Long Beach, CA
melonee.moses@dcma.mil

Using Inspections to Teach Requirements Validation

Lulu He, Dr. Jeffrey C. Carver, and Dr. Rayford B. Vaughn
Mississippi State University

Requirements validation is often not adequately covered by a traditional software engineering curriculum in universities. This article describes an experiment conducted in a graduate-level requirements engineering course to provide students a real world experience in requirements validation. The experiment made use of the N-fold inspection method, in which multiple teams of students inspect the same requirements document then meet together to discuss their findings. This procedure allows the students to not only practice their reviewing skills, but also to strengthen their communication and collaboration skills. At the conclusion of the exercise, the students were given the opportunity to provide qualitative and quantitative feedback. The results of this study suggest that the techniques employed by this class and the resulting defect detection could be useful in general during the requirements validation process.

It can be argued that requirements engineering (RE) is one of the most important stages in a traditional software development life cycle because it helps to correctly determine the desired purpose of a system. The goal of an RE process is to identify and document stakeholders and their needs in a form amenable to analysis, communication, and implementation [1]. Correct, complete, and unambiguous requirements not only ensure that developers build the right system, but also reduce the effort and cost that would otherwise be needed to fix requirements problems later. Because of the importance in obtaining correct requirements, RE courses are fundamental elements of any software engineering curriculum. An RE course should teach students techniques for accurately eliciting, analyzing, and validating requirements. By acquiring these skills, future software engineers are better equipped to produce quality requirements, eliminate faults, and reduce development time. Educating students in RE, however, is not an easy task because it is a human-centered process that requires skills from a variety of disciplines (e.g., computer science, system engineering, cognitive psychology, anthropology, and sociology) [1]. Moreover, RE educators must bridge the gap between academia and industry by exposing students to the real world as much as possible. One important real-world aspect of RE that has remained largely unstudied in the software engineering education community is requirements validation.

This article describes an exercise to aid in the teaching of requirements validation in a graduate-level RE course, along with an evaluation of its usefulness. In the exercise, the students validated a real software requirements specification document provided by an industrial partner using a meeting-based N-fold inspection method (described later) [2].

Background

Previous research has identified challenges faced during RE education. Because one goal of software engineering education is to help students develop the knowledge and skills necessary to be successful in industry, a challenge for RE educators is that the inherent complexity of industrial RE (i.e. the broad scope, multiple concerns, and deficient specifications) is difficult to replicate in a classroom environment [3, 4]. To be successful, a requirements engineer must possess both the technical skills needed to interact with the system and the social skills needed to interact with its human stakeholders [1]. These *soft skills* (e.g., communication and teamwork) are also difficult to teach in the classroom [5].

Most published work on RE education has focused on a small set of RE topics: elicitation, analysis, and the overall process. Validation is a complex task that is often not adequately covered in a traditional university education [6]. A requirements document can have many different types of deficiencies that may have disastrous effects on subsequent development stages and yield undesirable consequences [4, 7]. The requirements validation process checks for different types of problems (e.g., omissions, inconsistencies, and ambiguities) to help ensure that proper quality standards are fulfilled. Due to time limits and other considerations, validation is usually conducted informally either on an ad-hoc basis or simply as a peer review [8]. As a result, little has been reported concerning the educational challenges of validation topics like inspections. These challenges highlight the importance of improving education and critical, technical, and social skills that a requirements engineer must possess. However, the topics covered in many software engineering courses do not meet these needs, posing a major challenge for developing RE skills [6, 9]. Furthermore, experi-

ence reports about RE education is rare in software engineering and RE literature. To address the lack of focus on requirements validation, we developed a requirements validation exercise which we found to be successful and instructive. This exercise is a follow-on experiment from that reported at the Fourteenth Annual Systems and Software Technology Conference held in Salt Lake City in April 2002 titled “Third Party Walkthrough Inspections: A Joint Navy/University Empirical Software Engineering Project.” The most recent results of the experiment were also presented by invitation at the Canadian Air Force Software Engineering Symposium held at the Royal Military College of Canada in Kingston, Ontario, May 2007 [10, 11].

Description of the Requirements Validation Exercise

The exercise was conducted during a graduate-level RE course at Mississippi State University. The class included graduate students that were currently working for the Department of Defense (DoD), had previous government software development experience, and many that had no practical experience. The main goal of this course was to provide students with a comprehensive overview of requirements elicitation, analysis, specification validation, and management. These activities were introduced in the context of systems engineering and various software development life-cycle models. For each activity, the students were exposed to specific methods, tools, and notations. The semi-weekly, 75-minute class sessions contained a mixture of lecture material, class discussions centered on outside readings, and student presentations. Near the end of the semester, the students participated in a two-week requirements validation exercise – the primary subject of

this article. While the exercise reported here is based on a small number of students, we believe the results achieved suggest that such techniques could be considered in larger requirements validation exercises. The authors are amenable to cooperating with others to expand this research through additional empirical investigation – particularly in DoD software engineering endeavors.

Overview and Goals of the Requirements Validation Exercise

The overall goal of the exercise was to provide students with hands-on practice in requirements validation. The specific goals of the requirements validation exercise were the following:

1. To help students understand the course materials better by experiencing the format and presentation of a real requirements document, experiencing the impact of domain-specific language in understanding and reviewing a requirements document, and exposing the students to flaws commonly found in a requirements document.
2. To help students obtain an appreciation for the complexity and necessity of RE, especially requirements validation.
3. To give students hands-on experience validating a requirements document with specific inspection techniques.
4. To provide an opportunity for students to practice *soft skills* such as communication and teamwork.

Two important goals of the course were addressed through this exercise. First, give the students an idea of the size, complexity, language, and flaws that can occur in software artifacts, they validated a real requirements document, which was actually used by a contractor to develop and implement a system. The requirements document described an upgrade to a case-tracking system for the U.S. National Labor Relations Board. The 43-page document was written in natural language (English) and contained the standard content that would be expected in a government requirements document.

Second, to expose students to a real-world requirements validation method, the *N-fold inspection method* was used. In this method, the same artifact is inspected by multiple teams in parallel with the goal of improving the overall review effectiveness [12]. From an educational point of view, the N-fold inspection method provides students with an opportunity to discuss the defects found with other students, thereby understanding how others had viewed the artifact. Furthermore, N-fold inspection

meetings expose students to the importance of communication and teamwork – important soft skills.

Requirements Validation Techniques

Studies have shown that inspections are an effective requirements validation technique because they greatly improve system quality by detecting many defects early in the software life cycle [13]. Within the N-fold method, individual reviewers can use different approaches to review the document. Our students used either a checklist approach or the Perspective-Based Reading (PBR) approach (each described in more detail).

In practice, most industrial inspections use an ad-hoc or checklist-based approach for defect detection [14]. A checklist provides the inspector with concrete guidance that is not provided by an ad-hoc approach. A checklist is a list of items that focus an inspector's attention on specific topics, such as common defects or organizational rules, while reviewing a software

perspectives are the user, the designer, and the tester. PBR techniques provide reviewers with a set of steps to follow to build a high-level system abstraction and questions to help identify problems. For example, a reviewer assuming the user perspective may create use cases, while a reviewer assuming the tester perspective would create test plans. Studies have shown that PBR is a more effective, systematic, focused, goal-oriented, customizable, and transferable process than a standard checklist or ad-hoc approach [16]. By reviewing the requirements document from the perspective of different stakeholders (i.e., playing the role of that stakeholder), the reviewers are expected not only to detect more defects, but also to better comprehend the complexity of RE.

Details of the Requirements Validation Exercise and Summary of Research Results

Students enrolled in the course were divided into four, three-person teams with the expertise level balanced across teams. The members of two teams used a checklist to inspect the requirements document, while members of the other two teams used the PBR technique. For the PBR teams, each student was instructed to use one of the three perspectives (user, designer, or tester) to guide their inspection. These two inspection techniques were chosen because research suggested that while PBR is often a more effective technique, checklists are more widely used in government and industry [14]. A second motivator was that no previous work had compared the effectiveness of a checklist-based approach to the effectiveness PBR in the context of the N-fold inspection method. More importantly, instead of using a generic checklist, the checklist in this study was specifically developed for the type of requirements document to be inspected (e.g., from a government organization).

Before the exercise began, the students received one class meeting (75 minutes) of training in their assigned technique (checklist or PBR). The training for the checklist teams was done through a discussion with the course professor about attributes of requirements quality. During this discussion, the checklist students – heavily guided by the professor – developed a checklist to guide their inspection. The training for PBR was done by an expert in PBR and included a discussion of the theory behind the techniques and a case study that illustrated an example of its use. After the training, the

“Studies have shown that inspections are an effective requirements validation technique ... by detecting many defects early in the software life cycle.”

document [15]. For this requirements validation exercise, an informal checklist was developed that focused on important quality concepts relevant to a requirements document. Checklist examples are readily available on the Web, for example, <software.gsfc.nasa.gov/AssetsApproved/PA2.2.1.5.doc>, <www.processimpact.com/process_assets/requirements_review_checklist.doc>, or <www.swqual.com/training/Require.pdf>. For the class exercise, we used a checklist based on the Volere Requirements Specification Template which can be found at <www.systemsguild.com/GuildSite/Robs/Template.html>.

PBR is a systematic inspection technique that supports defect detection in software requirements through a role-playing exercise [13]. In PBR, each reviewer verifies the correctness of the requirements from the perspective of a specific stakeholder. The most common

PBR reviewers were given the detailed protocol for their assigned perspective. Then each student performed an individual inspection of the requirements document using their assigned technique. During this inspection, each student individually reviewed the document and recorded as many defects as he or she could find. The students were given two days to perform this task outside of class. Once all three members of a team completed the individual inspection step, they met together in the 1st Team Meeting. During this meeting, the students discussed the defects they found and agreed on a final team defect list. After all four teams had conducted the 1st Team Meeting, the two checklist teams (six students) met together and the two PBR teams (six students) met together for the 2nd Team Meeting. In these six-person meetings, the reviewers examined the two defect lists produced during the 1st Team Meeting and agreed on a final list of defects.

The data analysis indicated that PBR was more effective, both for individual inspectors and for the teams. Conversely, the data suggested that the checklist teams had more effective team meetings during the N-fold inspection process than the PBR teams. Here, the effectiveness of team meetings is defined by two measures: *meeting gains*, i.e. the number of defects identified during the meeting discussions that no individual inspector had found prior to the meeting, and *meeting losses*, i.e. defects found by an individual inspector that the inspection team determines are false positives and, hence, not recorded on the final team defect list. In the case of the PBR teams, the team meeting served little purpose because there were few meeting gains or meeting losses. The end result would have been similar had the individual list simply been combined without spending time in a formal team meeting. Conversely, for the checklist teams, the meeting served a vital role in the process. During the team meetings, not only were there meeting gains, but also a large percentage of false positives were identified and removed as meeting losses. Identification of false positives saves time during the rework phase. One likely cause of this result is the different perspectives from which the PBR reviewers approached the Software Requirements Specification (SRS). Each PBR reviewer focused on their own perspective and was less concerned with the perspectives of others. For the checklist team, the reviewers inspected the SRS using the same checklist and there was more interaction among team members

during the meetings. The most important, and novel, conclusion drawn from these results is that the effectiveness and necessity of a team meeting depends greatly on the technique used during the individual preparation phase of the inspection. Additional data on this experiment can be obtained by contacting the second or third author of this article.

Evaluation of the Educational Value of the Exercise

To evaluate the effectiveness of this exercise relative to the four educational goals listed earlier, quantitative and qualitative data was collected using a post-study survey. Goals 1-3 were specifically evaluated by the survey questions in Table 1 (see page 14). Goal 4 was addressed by using team meetings, but it was not specifically evaluated on the post-study questionnaire. The survey focused on the students' opinions of the exercise and gave them an opportunity to provide feedback on how to improve the exercise. The first two questions were answered using a predetermined scale (explained with each question). The other questions were answered using free text.

Results

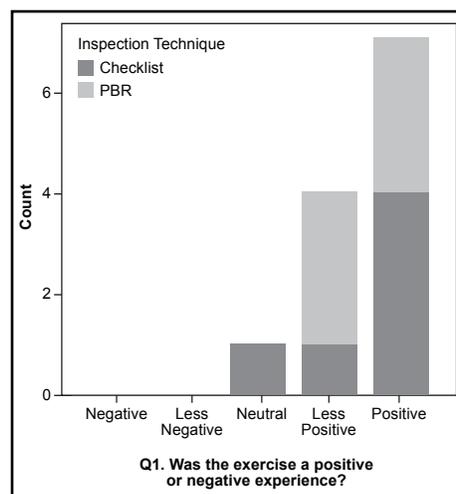
This section discusses the results from the post-exercise questionnaire along with a brief explanation. The questionnaire collected both qualitative and quantitative data from the students.

Quantitative Data

Q1. Was the exercise a positive or negative experience?

The students answered this question on a scale of 1 (negative) to 5 (positive). Figure 1 shows that 91 percent of the students found the exercise to be a positive experi-

Figure 1: Positive or Negative Experience Results



ence with no students having a negative experience. In Figure 1, the ratings given by students using PBR and checklist are shown in different shades to evaluate any impact the technique had. While the opinion of the PBR reviewers is slightly more positive, there is little difference due to the technique used. Therefore, regardless of which technique was used, the students found the N-fold inspection exercise to be a positive experience.

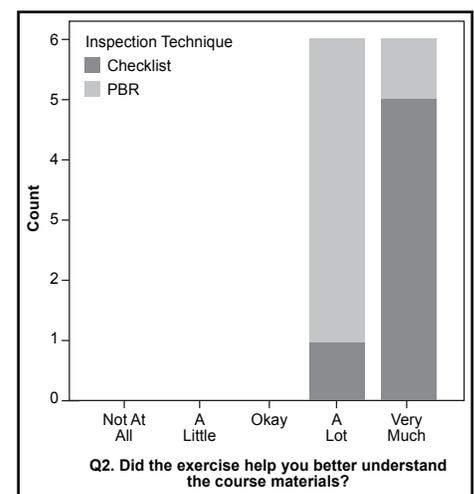
Q2. Did the exercise help you better understand the course materials?

The students responded to this question on a scale of 1 (not at all) to 5 (very much). Figure 2 shows that all of the students found the exercise was *very helpful* or a *lot* helpful.

Qualitative Data

The qualitative feedback, in which the students were able to express their own opinions, provides additional insight into the usefulness and effectiveness of the exercise. In this analysis, specific answers given for each question are listed along with the number of students who gave that answer, shown in parentheses where *3 checklist/2 PBR* means that three students who used the checklist and two students who used PBR gave the answer. In some cases, the response was given by only PBR students or only checklist students, e.g., *3 PBR* in bullet three of Q3. This does not imply that the answer is not applicable to the other technique – it only means that no students using the other technique provided an answer. In some cases, students provided more than one answer to each question, so the total number of responses may be greater than 12. Q3 and Q4 from the post-study survey were geared towards addressing Educational Goals 2 and 3. Q5

Figure 2: Understanding the Course Materials Results



Q1. Was the exercise a positive or negative experience?
Q2. Did the exercise help you better understand the course materials?
Q3. Did you see any benefits from the exercise? If so what were they?
Q4. Did you learn anything by performing the exercise? If so, what?
Q5. Did you see any drawbacks to the exercise? If so what were they?
Q6. How could the exercise been improved?

Table 1: *Survey Questions*

and Q6 provide feedback on how to improve the exercise.

Q3. Did you see any benefits from the exercise? If so what were they?

All 12 students indicated that they found some benefit from participation in this exercise:

1. It provided hands-on/practical experience (3 checklist/2 PBR).
2. It helped them better understand a real requirements document (3 checklist /3 PBR).
3. It helped them better understand defect detection in a requirements document (3 PBR).

Because the students obtained hands-on experience and indicated that they understood a real requirements document, this exercise helped address Educational Goals 2 and 3.

Q4. Did you learn anything by performing the exercise? If so, what?

From this exercise, the students learned the following:

1. The usefulness of inspections (2 checklist/1 PBR).
2. The difficulties involved in creating and using a real requirements document (3 checklist/1 PBR).
3. The benefits of using a method to focus a requirements inspection on certain aspects of the requirements document (2 checklist/5 PBR).

Similar to Q3, these responses indicate that the students learned the benefits of inspections, and the difficulties in creating real requirements documents, helping address Educational Goals 2 and 3.

Q5. Did you see any drawbacks to the exercise? If so what were they?

Only five (4 checklist/1 PBR) of the 12 students reported any drawbacks of the exercise:

1. Not enough training (3 checklist/1 PBR).
2. Not enough time (2 checklist).

On a positive note, these drawbacks all relate to the logistics of the exercise and not to its intrinsic value. None of these drawbacks are concerned with the inspec-

tion procedure that was used during the exercise.

Q6. How could the exercise have been improved?

1. Expand the exercise (1 checklist/1 PBR).
2. Provide more domain knowledge (2 checklist).
3. Allow more time for various activities (2 checklist/2 PBR).
4. Provide more training (1 checklist/4 PBR).

These suggestions generally relate to the drawbacks cited in Q5. It was interesting that two students asked for a more extensive exercise that allowed them to obtain more practice and experience using the inspection techniques. This request suggests that the students believed that even more benefit would be obtained by expanding the scope of the exercise.

Summary and Conclusion

This article describes the use of a requirements validation exercise in a graduate-level RE course. In the exercise, the students validated a software requirements document using a meeting-based N-fold inspection. The students provided their opinions and feedback about the usefulness of the exercise in a post-exercise survey. These results showed that the exercise achieved its goals.

Overall, students were highly satisfied with the exercise content and found it to be helpful. The exercise helped students understand what a software requirements document looks like and gain insight into the difficulty and complexity involved in its correct development. They not only realized the importance of requirements validation but also gained hands-on experience using inspection techniques. The exercise provided the students with essential knowledge about requirements quality, enabling them to better understand other course material such as the following: elicitation, analysis, and specification. Moreover, though not empirically evaluated, the team meetings in this exercise gave the students an opportunity to practice

their communication and teamwork skills, which are essential for their future careers.

Based on the feedback provided by the students, the following modifications are being considered for future similar class exercises:

1. Provide more training on the inspection techniques by using case studies.
2. Give specific instructions on the structure and organization of the team meeting.
3. Extend the length of the exercise to allow additional time for training and the individual inspection.
4. To make the exercise more realistic, stakeholders of the software requirements document will be invited to class, and help students to obtain more domain knowledge.
5. When the N-fold inspection is finished, the defect lists will be given to the owner of the requirements document to understand the disposition of those defects.

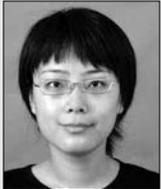
This study was performed in a graduate-level university course; therefore, the next step is to perform additional validation in an industrial setting. Furthermore, the positive experience with the N-fold inspection process during RE suggests that it may have applicability in other phases of the software life cycle. In the future, we will seek out opportunities to replicate these experiences in other aspects of the software engineering curriculum. ♦

References

1. Nuseibeh, B., and S. Easterbrook. "Requirements Engineering: A Road Map." Proc. of International Conference on Software Engineering, Limerick, Ireland, June 2000: Association of Computing Machinery (ACM) Press, 2000: 37-46.
2. He, L., and J.C. Carver. "PBR Vs. Checklist: A Replication in the N-Fold Inspection Context." Proc. of 5th ACM/Institute of Electrical and Electronics Engineers (IEEE) International Symposium on Empirical Software Engineering (ISESE 2006), Rio de Janeiro, Brazil, Sept. 21-22, 2006.
3. Armarego, J., and S. Clarke. "Preparing Students for the Future: Learning Creative Software Development – Setting the Stage." Proc. of the Annual International Higher Education Research and Development Society of Austral Asia Conference Christchurch, New Zealand. July 6-9, 2003.
4. Van Lamsweerde, A. "Requirements Engineering in the Year 00: A Research Perspective." Proc. of 22nd

- International Conference on Software Engineering. Limerick, Ireland, 2000: IEEE Computer Society Press, 2000.
5. Conn, R. "Developing Software Engineers at the C-130j Software Factory." *IEEE Software* 19.5 (2002): 25-29.
 6. Bubenko, J.A. "Challenges in Requirements Engineering." Proc. of 2nd IEEE International Symposium on Requirements Engineering, Los Alamitos, CA: IEEE Computer Society, 1995: 160-162.
 7. Meyer, B. On Formalism in Specifications. *IEEE Software* 2.1 (1985): 6-26.
 8. Rosca, D. "An Active/Collaborative Approach in Teaching Requirements Engineering." Proc. of 30th Annual Frontiers in Education Conference, Kansas City, MO., Oct., 2000: 9-12.
 9. Lethbridge, T.C. "What Knowledge Is Important to a Software Professional?" *Computer* 33.5 (2000): 44-50.
 10. Vaughn, R.B., and J. Lever. "Third Party Walkthrough Inspections: A Joint Navy/University Empirical Software Engineering Project." Proc. of Fourteenth Annual Systems and Software Technology Conference, Salt Lake City, UT, Apr. 29-May 2, 2002.
 11. Vaughn, R.B., and J.C. Carver. "Experiences in N-Fold Structured Walkthroughs of Requirements Documents." Proc. of Canadian Air Force Software Engineering Symposium., Royal Military College of Canada, Kingston, Ontario. 24-25 May, 2007.
 12. Martin, J., and W. Tsai. "N-Fold Inspection: A Requirements Analysis Technique." *Communications of the ACM* 33.2 (1990): 223-232.
 13. Basili, V.R., et al. "The Empirical Investigation of Perspective-Based Reading." *Empirical Software Engineering: An International Journal* 1.2 (1996): 133-164.
 14. Laitenberger, O., K.E. Emam, and T.G. Harbich. "An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-Based Reading of Code Documents." *IEEE Transactions on Software Engineering* 27.5 (2001): 387-421.
 15. Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." *IBM System Journal* 15.3 (1976): 182-211.
 16. Shull, F., I. Rus, and V.R. Basili. "How Perspective-Based Reading Can Improve Requirements Inspection." *IEEE Software* 33.7 (2000):73-79.

About the Authors



Lulu He is a doctorate student in the Computer Science and Engineering Department at Mississippi State University. She received her bachelor's and master's degrees in computer science from Wuhan University, China in 2001 and 2004, respectively. She also received a master's degree in computer science from Mississippi State University in August, 2007. Her research interests include Software Quality, Software Inspections, Software Architecture, and Software Engineering for Scientific and Engineering Computing.

**Computer Science
and Engineering
PO Box 9637
Mississippi State University
Mississippi State, MS 39762
Phone: (662) 325-8798
E-mail: lh221@msstate.edu**



Jeffrey Carver, Ph.D., is an Assistant Professor in the Computer Science and Engineering Department at Mississippi State University. He received his doctorate degree from the University of Maryland in 2003. His research interests include software process improvement, software quality, software inspections, and software engineering for scientific and engineering computing. He has more than 30 refereed publications in these areas. His research has been funded by the U.S. Army Corps of Engineers, the U.S. Air Force, and the National Science Foundation.

**Computer Science
and Engineering
PO Box 9637
Mississippi State University
Mississippi State, MS 39762
Phone: (662) 325-0004
Fax: (662) 325-8997
E-mail: carver@cse.msstate.edu**



Rayford B. Vaughn, Ph.D., received his doctorate from Kansas State University in 1988. He is a William L. Giles Distinguished Professor and the Billie J. Ball Professor of Computer Science and Engineering at Mississippi State University and teaches and conducts research in the areas of Software Engineering and Information Security. Prior to joining the university, he completed a 26-year career in the U.S. Army retiring as a Colonel and three years as a Vice President of Defense Information Systems Agency Integration Services, and EDS Government Systems. Vaughn has more than 100 publications to his credit and is an active contributor to software engineering and information security conferences and journals. In 2004, he was named a Mississippi State University Eminent Scholar. Vaughn is the current Director of the Mississippi State University Center for Critical Infrastructure Protection and the Center for Computer Security Research.

**Computer Science
and Engineering
PO Box 9637
Mississippi State University
Mississippi State, MS 39762
Phone: (662) 325-7450
Fax: (662) 325-8997
E-mail: vaughn@cse.msstate.edu**

Integrating Software Assurance Knowledge Into Conventional Curricula

Dr. Nancy R. Mead
Software Engineering Institute

Dr. Dan Shoemaker
University of Detroit Mercy

Jeffrey A. Ingalsbe
Ford Motor Co.

One of our challenges is deciding how best to address software assurance in university curricula. One approach is to incorporate software assurance knowledge areas into conventional computing curricula. In this article, we discuss the results of a comparison of the Common Body of Knowledge (CBK) for Secure Software Assurance with traditional computing disciplines. The comparison indicates that software engineering is probably the best fit for such knowledge areas, although there is overlap with other computing curricula as well.

Defects are not an option in today's world. Much of our national well-being depends on software. So the one thing that America's citizens should be able to expect is that that software will be free of bugs. Sadly, that is not the case. Instead, *commonly used software engineering practices permit dangerous defects that let attackers compromise millions of computers every year*. That happens because *commercial software engineering lacks the rigorous controls needed to (ensure defect free) products at acceptable cost* [1].

Most defects arise from program or design flaws, and they do not have to be actively exploited to be considered a threat [2, 3]. In fiscal terms, the exploitation of such defects costs the American economy an average of \$60 billion dollars a year [4]. Worse, it is estimated that *in the future, the nation may face even more challenging problems as adversaries – both foreign and domestic – become increasingly sophisticated in their ability to insert malicious code into critical software systems* [3].

Given that situation, the most important concern of all might be that the exploitation of a software flaw in a basic infrastructure component such as power or communication could lead to a significant national disaster [5]. The Critical Infrastructure Taskforce sums up the likelihood of just such an event in the following:

The nation's economy is increasingly dependent on cyberspace. This has introduced unknown interdependencies and single points of failure. A digital disaster strikes some enterprise every day, [and] infrastructure disruptions have cascading impacts, multiplying their cyber and physical effects. [5]

Predictions such as this are what motivated the National Strategy to Secure Cyberspace, which mandates the Department of Homeland Security (DHS) to do the following:

... promulgate best practices and methodologies that promote

integrity, security, and reliability in software code development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development. [5]

Given the scope of that directive, one obvious solution is to ensure that secure software practices are embedded in workforce education, training, and development programs nationwide. The problem is that there is currently no authoritative point of reference to define what should be taught [3]. For that reason, in 2005 DHS created a working group to define a CBK for Secure Software Assurance <<https://buildsecurityin.us-cert.gov/daisy/bsi/resources/dhs/95.html>>. The goal of the CBK is to itemize all of the activities that might be involved in producing secure code. DHS does not intend the CBK to be used as a general *standard, directive, or policy* [3]. Instead, its sole purpose is to catalog secure practices that might be appropriate to currently existing academic disciplines. Thus, the CBK is an inventory of potential *knowledge areas* within each contributing discipline.

The CBK assumes the following:

... software assurance is not a separate profession. What is not clear, however, is the precise relationship between the elements of the CBK and the curricula of each potential relevant field. [6]

So, the challenge is to correctly integrate secure software assurance practices into each contributing discipline [3, 6].

Several disciplines could conceivably benefit from CBK, such as *software engineering, systems engineering, information systems security engineering, safety, security, testing, information assurance, and project management* [3]. Consequently, in order to ensure that the right content is taught in each, it is necessary to understand the proper relationship

between the CBK and the curricula of each relevant discipline [6].

Finding Where the CBK Fits Into Current Curricula

The overall goal of the CBK is to *ensure adequate coverage of requisite knowledge areas in each contributing discipline* [3]. Accordingly, the working group sought to understand the exact relationship of CBK elements to each traditional curriculum. Once that relationship was better understood, it was felt that it should be possible to recommend the right way to incorporate CBK content into each of the established disciplines.

That comparison was materially aided by the fact that the sponsoring societies of the three most influential academic studies had just finished their own survey of curricular models for computing curricula. This was reported in "Computing Curricula 2005: The Overview Report" commonly called CC2005 [7]. CC2005 merges the recommendations for the content and focus of *computer engineering, computer science, information systems, information technology, and software engineering* curricula into a single authoritative summary, which is fully endorsed by the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE) Computer Society, and the Association for Information Systems.

CC2005 specifies 40 topic areas. These 40 topics represent the entire range of subject matter for all five major computing disciplines. The report specifically states the following:

Each one of the five discipline-specific curricula represents the best judgment of the relevant professional, scientific, and educational associations and serves as a definition of what these degree programs should be and do. [7]

In addition to the 40 topic areas, which in effect capture all of the knowledge requirements for computing curricula

along with a ranking of their relative emphasis in each specific discipline, CC2005 also *summarizes the expectations for the student after graduation* [7]. This summary identifies 60 *competencies* that should be expected for each graduate. By referencing those identified competency outcomes, it is relatively easy to see the relationship between CBK knowledge elements and the CC2005 curricular requirements. It is also easier to see the places where there is a misalignment between the CBK and each discipline's curricular goals.

The CBK was mapped to the CC2005 recommendations for only three of the five disciplines. The two disciplines at opposite ends of the CC2005 continuum, *computer engineering* and *information technology*, were omitted because the former overlaps too much with electrical engineering and the latter overlaps too much with business.

Because these three curricula (computer science, information systems, and software engineering) have differing focuses, the content of CC2005 was examined one discipline at a time. First, a topic-by-topic analysis of *depth of coverage* was done. Depth of coverage was defined as *the quantity of material in the CBK that provides specific advice about how to execute a given activity in CC2005 in a more secure fashion*.

To obtain a metric, the assessment of *quantity of material* was based on a count of the textual references in the CBK that could be associated with each of the 40 topics. The assumption was that the more references to the topic in the CBK the greater the importance of integrating secure software assurance content into the teaching of that topic (see Table 1 for degree to which CC2005 Knowledge Areas are reflected in the CBK).

What Does This Mean?

The following eight CC2005 topic areas had a *significant* degree of coverage in the CBK (greater than 100 references): 1) requirements, 2) architecture, 3) design, 4) verification and validation (V&V), 5) evolution (e.g., maintenance), 6) processes, 7) quality, and 8) information systems project management. The following three CC2005 topic areas had *moderate* coverage in the CBK (less than 100 but more than 10): 1) legal/professional/ethics/society, 2) risk management, and 3) theory of programming languages.

This mapping shows that the main focus of the CBK is on generic software *work* rather than on the specific curricular aspects that characterize the study itself, such as algorithms (for computer science), or information systems management (for information systems). That indicates that

Knowledge Areas All Disciplines	Cites	Knowledge Areas All Disciplines	Cites
Integrative Programming (integrated)	9	Analysis of Requirements	1
Algorithms	1	Technical Requirements Analysis	274
Complexity	6	Engineering Economics for Software	1
Architecture	150	Software Modeling and Analysis	2
Operating Systems Principles and Design	5	Software Design	255
Operating Systems Configuration and Use	5	Software V&V	401
Platform Technologies	3	Software Evolution (Maintenance)	438
Theory of Programming Languages	10	Software Process	296
Human-Computer Interaction (HCI)	5	Software Quality	163
Graphics and Visualization	1		
Information Management (DB) Practice	1	Non-Computing Topics	Cites
Legal/Professional/Ethics/Society	93	Risk Management	86
Information Systems Development	7	Project Management	156

Table 1: *Degree to Which CC 2005 Knowledge Areas Are Reflected in the CBK*

CBK content would be best integrated into the places where the practical elements of the life cycle are introduced, such as a software design project course.

Fit Between the CBK and Desired Outcomes for the Profession

There were six priorities in CC2005. These range from *highest possible expectations* through *highest expectations* to *moderate expectations*, *low expectations*, *little expectations*, and *no expectations*. One of the more interesting aspects of CC2005 is the 60 expected competencies. Because there is a difference in focus for each discipline, there is a difference in what should be expected for each of them. For instance, there is a different set of presumed competencies for a computer scientist than for a software engineer.

The 60 expected competencies were taken directly from the 40 learning topics. Each competency was examined to determine which of the 40 topics could be assigned to it. For instance, if the competency was to *design a user-friendly interface*, there are 255 references in the CBK to *design*, and five references in the CBK to *human/computer interfaces*. So the number of CBK references for this outcome was assigned as 260 (Tables 2-4, pages 18-19).

For *computer science* there is a weak match between the CBK and *the highest possible competency expectations* in that only one of the eight outcomes (12.5 percent) had any degree of coverage. There is a slightly better match for the high expectations category,

three of 10 (30 percent). However, there is an excellent match with moderate expectations, nine of 12 (75 percent).

For *information systems* there is a reasonable match between the CBK and the *highest possible competency expectations* in that nine of the 22 competencies (40.9 percent) specified for that discipline are covered. There is no match for the high expectations category. However, there is a good match with moderate expectations, five of nine (55.5 percent).

For *software engineering* there is a strong match between the CBK and the *highest possible set of competency expectations* in that four of the seven outcomes (57.1 percent) are covered. There is reasonable match for the high expectations category in that three of 12 competencies are covered (25 percent). There is also a good match with moderate expectations in that five of nine areas are covered (55.5 percent).

Integrating the CBK into the World of Practical Education

One of the main inferences that can be drawn from this comparison is that the current CBK is less focused on theory than it is on application of the knowledge in practice. In essence, the results demonstrate that the CBK is built around and encapsulates knowledge about practical processes that are universally applicable to securing software rather than on discipline-specific concepts, theories, or activities.

This is best illustrated by the matches themselves. *Outcomes such as solve programming*

Computer Science		
Highest Possible Expectation (5)	Depth of Coverage	Conclusion
Solve programming problems (algorithms)	Analysis (274), Design (255)	strong
High Expectation (4)	Depth of Coverage	Conclusion
Do large-scale programming (programming)	Analysis (274), Design (255), Architecture (150)	strong
Develop new software systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
Create a software user interface (HCI)	HCI (5)	weak
Moderate Expectation (3)	Depth of Coverage	Conclusion
Create safety-critical systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
	Process (296), V&V (401), PM (156)	
Design information systems (IS)	Analysis (274), Design (255), Architecture (150)	strong
Implement information systems (IS)	Process (296), V&V (401), PM (156)	strong
Maintain and modify information systems (IS)	Evolution (438)	strong
Install/upgrade computers (planning)	Process (296), V&V (401), PM (156)	strong
Install/upgrade computer software (planning)	Process (296), V&V (401), PM (156)	strong
Design network configuration (networks)	Analysis (274), Design (255), Architecture (150)	strong
Manage computer networks (networks)	Evolution (438)	strong
Implement mobile computing system (networks)	Analysis (274), Design (255), Architecture (150)	strong

Table 2: Match Between CBK and CC2005 Expected Competencies for Computer Science

problems, do large scale programming, and design and develop new software and/or IS are high priorities in all of the disciplines. At the same time, each of these also has a significant degree of coverage in the CBK.

High-priority items in each of these disciplines that were not good matches with

the CBK tended to be such competencies as *prove theoretical results* (computer science), *develop proof-of-concept programs* (computer science), *select database products* (information systems), *use spreadsheet features well* (information systems), *do small scale programming* (software engineering), and *produce graphics or*

game software (software engineering).

Others, such as *create a software user interface*, were a mixed bag, with a good match to design but a poor match to HCI.

So, while these competencies might be individually important to their specific disciplines, they are not essential elements of

Table 3: Match Between CBK and CC2005 Expected Competencies for Information Systems

Information Systems/Information Technology		
Highest Possible Expectation (5)	Depth of Coverage	Conclusion
Create a software user interface (IS)	HCI (5)	weak
Define information system requirements (IS)	IS Dev. (7), Bus. Req. (1), Analysis (274)	strong
Design information systems (IS)	Design (255), Modeling (5), Architecture (150)	strong
Maintain and modify information systems (IS)	Evolution (438)	strong
Model and design a database (DB)	DB (1), Design (255)	strong
Manage databases (DB)	Evolution (438)	strong
Develop corporate information plan (planning)	Process (296)	strong
Develop computer resource plan (planning)	PM (156)	strong
Schedule/budget resource upgrades (planning)	PM (156)	strong
Develop business solutions (integration)	Bus. Req (1), Modeling (5), Design (255)	strong
High Expectation (4)	Depth of Coverage	Conclusion
None	n/a	
Moderate Expectation (3)	Depth of Coverage	Conclusion
Develop new software systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
Install/upgrade computer software (planning)	PM (155)	strong
Manage computer networks (networks)	Evolution (238)	strong
Manage communication resources (networks)	PM (155), Economics (1)	strong

Software Engineering		
Highest Possible Expectation (5)	Depth of Coverage	Conclusion
Do large-scale programming (programming)	Analysis (274), Design (255), Architecture (150)	strong
Develop new software systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
Create safety-critical systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
	Process (296), V&V (401), PM (156)	strong
Manage safety-critical projects (programming)	V&V (401), PM (156), Evolution (438)	strong
High Expectation (4)	Depth of Coverage	Conclusion
Develop new software systems (programming)	Analysis (274), Design (255), Architecture (150)	strong
Create a software user interface (HCI)	HCI (5)	weak
Define information system requirements (IS)	IS Development (7), Bus. Req. (1), Analysis (274)	strong
Moderate Expectation (3)	Depth of Coverage	Conclusion
Design a human-friendly device (HCI)	HCI (5), Design (255)	strong
Design information systems (IS)	Design (255), Modeling (5), Architecture (150)	strong
Maintain and modify information systems (IS)	Evolution (438)	strong
Install/upgrade computers (planning)	Process (296), V&V (401), PM (156)	strong
Install/upgrade computer software (planning)	Process (296), V&V (401), PM (156)	strong
Manage computer networks (networks)	Evolution (438)	strong
Implement mobile computing system (networks)	Analysis (274), Design (255), Architecture (150)	strong

Table 4: Match Between CBK and CC2005 Expected Competencies for Software Engineering

secure software assurance as defined by the CBK. In view of that finding, it would appear to be easier to introduce CBK content into curricula that is focused on teaching pragmatic software processes and methods. And given its historic involvement with those areas, the discipline of software engineering might be the place to start.

Therefore, one additional suggestion might be that a similar study should be done based strictly on Software Engineering 2004, "Curricular Guidelines for Undergraduate Programs in Software Engineering," particularly Table 1: Software Engineering Education Knowledge Elements [8]. That itemizes a set of *knowledge areas and knowledge units* that are similar in focus and purpose to the 40 knowledge areas contained in CC2005. Thus, it should be possible to better understand the actual relationship between standard software engineering curricular content and the contents of the CBK through that comparison.

There is another distinct observation arising out of this study. Although the *moderate expectations* category does not reflect priority areas, it is overwhelmingly the best aligned category for *each* discipline. What that might indicate is that, although secure software assurance is a legitimate area of study for all of these fields, it is not the highest priority in any of them. In terms of disciplinary implementations, the practitioner orientation and the fact that security

content is not the point of these fields indicates that courses that cover practical life-cycle functions might be the place to introduce secure software assurance content within any given discipline.

As a final note, the measurement process used in this study (e.g., a raw count) is inherently less accurate than expert contextual analysis of the meaning of each knowledge element. Therefore, a more rigorous comparison should be undertaken to better characterize the functional relationship between the items in the CBK and the various curricular standards. This would be particularly justified for the study of software engineering curricular content mentioned above. Once a means of comparison that everybody can agree on is used, it should be relatively simple to work out the nuts-and-bolts of specific implementations within each individual program. ♦

References

1. President's Information Technology Advisory Committee. Cybersecurity: A Crisis of Prioritization. Arlington: Executive Office of the President, National Coordination Office for Information Technology Research and Development, 2005.
2. Jones, Capers. Software Quality in 2005: A Survey of the State of the Art. Marlborough: Software Productivity Research, 2005.
3. Redwine, Samuel T., Ed. Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software, Version 1.1. Washington: U.S. DHS, 2006.
4. Newman, Michael. Software Errors Cost U.S. Economy \$59.5 Billion Annually. Gaithersburg: National Institute of Standards and Technology (NIST), 2002.
5. Clark, Richard A., and Howard A. Schmidt. A National Strategy to Secure Cyberspace. Washington: The President's Critical Infrastructure Protection Board, 2002 <www.us-cert.gov/reading_room/cyberspace_strategy.pdf>.
6. Shoemaker, D., A. Drommi, J. Ingalsbe, and N.R. Mead. "A Comparison of the Software Assurance Common Body of Knowledge to Common Curricular Standards." Dublin: 20th Conference on Software Engineering Education and Training, 2007.
7. Joint Taskforce for Computing Curricula. Computing Curricula 2005: The Overview Report. ACM/AIS/IEEE, 2005.
8. Joint Taskforce for Computing Curricula. Software Engineering 2004, Curricular Guidelines for Undergraduate Programs in Software Engineering. ACM/IEEE, 2004.

About the Authors



Nancy R. Mead, Ph.D., is a senior member of the technical staff in the Networked Systems Survivability Program at the SEI. She is also a faculty member at Carnegie Mellon University. Mead's research interests are in the areas of information security, software requirements engineering, and software architectures. She is a Fellow of the IEEE and the IEEE Computer Society and is also a member of the Association for Computing Machinery. Mead received her doctorate in mathematics from the Polytechnic Institute of New York, and received bachelor's and master's degrees in mathematics from New York University.

SEI
4500 5th AVE
Pittsburgh, PA 15213
E-mail: nrm@sei.cmu.edu



Dan Shoemaker, Ph.D., is the director of the Centre for Assurance Studies. He has been professor and chair of computer and information systems at the University of Detroit Mercy for 24 years, and co-authored the textbook, "Information Assurance for the Enterprise." His research interests are in the areas of secure software assurance, information assurance and enterprise security architectures, and information technology governance and control. Shoemaker has both a bachelor's and a doctorate degree from the University of Michigan, and master's degrees from Eastern Michigan University.

Computer and Information Systems – College of Business Administration
University of Detroit Mercy
Detroit, MI 48221
Phone: (313) 993-1202
E-mail: shoemadp@udmercy.edu



Jeffrey A. Ingalsbe is a senior security and controls engineer with Ford Motor Company where he is involved in information security solutions for the enterprise, threat modeling efforts, and strategic security research. He has a bachelor's degree in electrical engineering and a master of science degree in computer information systems from Michigan Technological University and the University of Detroit Mercy, respectively. He is currently working on a doctorate in software engineering at Oakland University. Ingalsbe serves as an expert industry panelist on two national working groups within the DHS's Cybersecurity Division.

17475 Federal DR
STE 800-D04
Allen Park, MI 48101
Phone: (313) 390-9278
E-mail: jingalsb@ford.com



Systems & Software Technology Conference

"Technology: Tipping the Balance"

To explore new and needed technologies, as well as lessons learned, which tip the balance in the favor of our Defense Services, providing them an asymmetric advantage.

TOPICS COVERED

Assurance and Security
Estimating and Measuring
Lessons Learned
New Concepts and Trends
Policy and Standards
Processes and Methods
Professional Development
Robust Engineering
Systems: from Design to Delivery

20th Annual Systems & Software Technology Conference (SSTC 2008)
29 April – 2 May 2008 Las Vegas Hilton Hotel Las Vegas, Nevada

REGISTER TODAY!



WHO SHOULD ATTEND

Acquisition Professionals • Program/Project Managers
Programmers • System Developers
Systems Engineers • Process Engineers
Quality and Test Engineers

Complete conference schedule and registration information available 7 January 2008
visit: www.sstc-online.org

Software Engineering Continuing Education at a Price You Can Afford

Maj Christopher Bohn, Ph.D.
Air Force Institute of Technology

Software is so critical to today's military programs that failure of the software generally results in failure of the program. Hoping for success is the surest way to avoid it. Avoiding software failure is not accidental; it requires careful application of sound software engineering. Whether you are an engineer, a program manager, a contracting officer, or anyone else involved in the development, acquisition, or sustainment of a software-intensive system, the Air Force Institute of Technology's (AFIT) Software Professional Development Program (SPDP) can bring you the education you need to achieve software success¹.

Software is everywhere. It is in our kitchen Appliances; it is in our cars. It allows us to communicate worldwide, and it helps us manage our personnel systems. It is in our weapons systems. Famously, 80 percent of the F-22's functionality is performed in software; some have said that taking a picture of an F-22 is the only thing you can do with it that does not require software [1]. Software is so integral to the F-35 that Lockheed Martin spearheaded the effort to create a safety-critical C++ standard [2]. There is no project in today's military that is not affected by software. Software, like any other technology, has its limitations; as dependent as we have become on software, those limitations become our limitations. Many wonder how we can add armor to our programs' Achilles heels.

Perhaps you were motivated by Secretary Wynne's emphasis on making education a priority in your career [3, 4]. Perhaps you read the National Defense Industrial Association's report on the top defense software engineering issues and are wondering how you can overcome these issues in your program [5]. Perhaps you are simply looking for some job-relevant education to satisfy the Acquisition Professional Development Program continuing education requirements without taxing your unit's travel budget. We are here to help you. We can bring education to your office or home, and we will not charge you or your organization a dime.

The AFIT's SPDP is a distance learning, professional continuing education program designed to benefit the Department of Defense (DoD) organizations and individuals with varying levels of experience and responsibility. SPDP is well into its second decade, but it has been anything but stagnant; we have constantly been adapting to meet the needs of the defense software engineering community. You may have read about SPDP when it transitioned from a resident program to satellite-delivered distance learning program [6, 7]. You may have read about SPDP when it transitioned from satellite delivery to Internet streaming and

from quarter-long courses to month-long courses [8]. Since then, we have been working to improve the education we provide to our students [9, 10].

A typical SPDP course is four weeks long; the lectures are available through Internet streaming or they can be downloaded to view offline. This format has permitted our students to complete courses in a high-paced environment; we have even had students take SPDP courses while deployed

“Software, like any other technology, has its limitations; as dependent as we have become on software, those limitations become our limitations.”

to Southwest Asia and while at sea. SPDP courses differ from asynchronous Web-based training in many ways. The most significant is that they are instructor-led courses – real, human instructors who provide the lectures (see Figure 1, page 22), complemented with reading assignments from a textbook. We also make use of online discussion boards and sometimes teleconferences to provide continuous instructor-student and student-student interaction. Finally, our students are evaluated with homework assignments and exams.

Available Courses

We currently offer 12 SPDP courses. We have two courses focused on project management:

- CSE 479, Software Project Initiating and Planning.
- CSE 480, Software Project Monitoring and Control.

The software engineering life cycle is cov-

ered in six courses. CSE 481, Introduction to Software Engineering, provides an overview, and each major activity of the software life cycle is covered in greater detail in its own course:

- CSE 482, Software Requirements.
- CSE 483, Software Design.
- CSE 484, Software Implementation.
- CSE 485, Software Systems Maintenance.
- CSE 486, Verification, Validation and Testing.

Our next three courses address object-oriented development:

- CSE 487, Fundamentals of Object-Oriented Systems.
- CSE 488, Modeling Object-Oriented Systems using UML.
- CSE 489, Advanced Analysis and Design of Object-Oriented Systems.

Our final course, CSE 496, Software Engineering Practicum, is a three-week resident course held at our campus near Wright-Patterson AFB, Ohio.

With the exception of CSE 489 and CSE 496, none of these courses have prerequisites. You can choose to take them all or only the ones you are interested in and you can take them in any order. We have committed to offering each of these courses at least once per year, though when demand and faculty resources are in accord, we will provide more frequent offerings.

As stated earlier, a typical SPDP course is four weeks long. Each week, the instructor will provide two lectures and perhaps will hold an optional teleconference. There will be reading assignments and homework assignments. There may be a mid-term exam, and the class will conclude with a final exam.

The atypical courses are CSE 489 and CSE 496, as these are project-based courses. CSE 489 is still a four-week, instructor-led course, but there is only one lecture per week and one mandatory teleconference in which the students present project progress. Enrollment in CSE 489 requires completion of CSE 487 and CSE 488. CSE 496 is a three-week resident course in which stu-



Figure 1: The author teaches CSE 489 while attending the 2006 Systems and Software Technology Conference.

dents form a development team and have the opportunity to apply what they have learned; before enrolling in CSE 496, a student must have completed at least seven other SPDP courses.

Certifications

AFIT offers four certifications to help you mark your progress through SPDP and to help you demonstrate that you have taken a breadth of courses. The Software Engineering Management Certificate is awarded in recognition of successful completion of topics related to software project management and the individual components of the software life-cycle model; it will be awarded for the completion of CSE 479, CSE 480, and CSE 481. The Software Lifecycle Development Certificate is awarded in recognition of a more in-depth study of each of the phases of the software life cycle; it will be awarded after the successful completion of CSE 482, CSE 483, CSE 484, and CSE 485. The Advanced Software Development Certificate is awarded after the successful completion of CSE 486, CSE 487, and CSE 488, in recognition of these analysis, modeling, and testing topics. Finally, the Technical Software Development Certificate is awarded after the successful completion of CSE 489 and CSE 496, in recognition of completing the project-centered courses in the program.

Besides the AFIT certifications, SPDP can help you toward another credential. AFIT's School of Systems and Logistics is one of seven registered educational providers worldwide for the Institute of Electrical and Electronics Engineers (IEEE) Computer Society's Certified Software Development Professional (CSDP) program [11]. The CSDP certification is the only software development certification that has all of the components of a professional certification: an exam demon-

strating mastery of the Software Engineering Body of Knowledge (SWE-BOK), an experience base, and continuing education. While taking the SPDP courses is neither sufficient nor necessary to earn the CSDP, completing the curriculum will fully immerse you in the SWEBOK, preparing you for the CSDP exam.

Eligibility

SPDP classes are funded through AFIT for all DoD employees (active duty and reserve component service members and government civilians). Contractors and employees of other US. Government Agencies may also enroll in SPDP courses on a space-available basis. There is no tuition charged for SPDP courses, and AFIT provides textbooks free to DoD employees. Contractors and non-DoD government employees are responsible for procuring their own textbooks prior to the beginning of a course offering.

Nobody wants to be another statistic for the next study of defense software crises. You can mitigate this risk by arming yourself with software engineering knowledge and skills. Fortunately, with the SPDP, you will not need to dig into your tight travel funds and you will not need to figure out how to pay for expensive continuing education courses. Our faculty is here to help. The feedback we have received from our students and their supervisors is that they usually see improvement during their current projects.

If you are interested in taking SPDP courses, or at least curious, please visit the SPDP Web site at <www.afit.edu/ls/spdp/>, e-mail the faculty at <spdp@afit.edu>, or contact Candace Barker at (937) 255-7777 ext. 3319, or DSN 785-7777 ext. 3319.◆

Note

1. The views expressed in this article are those of the author and do not reflect the official policy or position of the Air Force, DoD, or the U.S. Government.

References

1. Ferguson, Jack. "Crouching Dragon, Hidden Software: Software in DoD Weapon Systems." *IEEE Software* July/Aug. (2001): 105-107.
2. Carroll, Kevin. "Deploying C++ for Use in International Safety-Critical Applications." Proc. from Systems and Software Technology Conference, 2007.
3. Wynne, Michael W. "Letter to Airmen: Education and the Airman." 13 Apr. 2006. <www.af.mil/library/viewpoints/secaf.asp?id=229>.

4. "Airman's Roll Call: Education Benefits Essential to Professional, Personal Development." 8 Aug. 2007 <www.af.mil/shared/media/document/AFD-070807-058.pdf>.
5. National Defense Industrial Association. "Top Software Engineering Issues within Department of Defense and Defense Industry." 2006. <www.ndia.org/Content/ContentGroups/Divisions1/Systems_Engineering/PDFs18/NDIA_Top_SW_Issues_2006_Report_v5a_final.pdf>.
6. "The Air Force Software Professional Development Program." *CROSSTALK* Dec. 1994.
7. "Distance Learning in the Air Force Professional Development Program: An Update." *CROSSTALK* Feb. 1995.
8. Hermann, Brian. "The AFIT Offers Software Continuing Education at Your Location at No Cost." *CROSSTALK* Dec. 2002.
9. Reisner, John. "Bridging the Gap: Peer-to-Peer Learning in a Distance Environment." Proc. Interservice/Industry Training, Simulation and Education Conference, 2004.
10. Bohn, Christopher A. "Watch Mr. Software." Proc. International Conference on Frontiers in Education: Computer Science and Computer Engineering.
11. *IEEE Computer Society* <www.computer.org/certification/>.

About the Author



Maj Christopher Bohn, Ph.D., is a Software Engineering Course Director at AFIT's School of Systems and Logistics,

where he teaches a series of distance-learning short courses. Over the past 14 years, he has served in various Air Force operational and research assignments. He is an IEEE-certified software development professional. Bohn has a bachelor's degree in electrical engineering from Purdue University, a master's degree in computer engineering from AFIT, and a doctorate from The Ohio State University.

AFIT/LS Research Park Campus
3100 Research BLVD
Kettering, OH 45420-4022
Phone: (937) 255-7777 ext. 3415
Fax: (937) 656-4654
E-mail: christopher.bohn@afit.edu

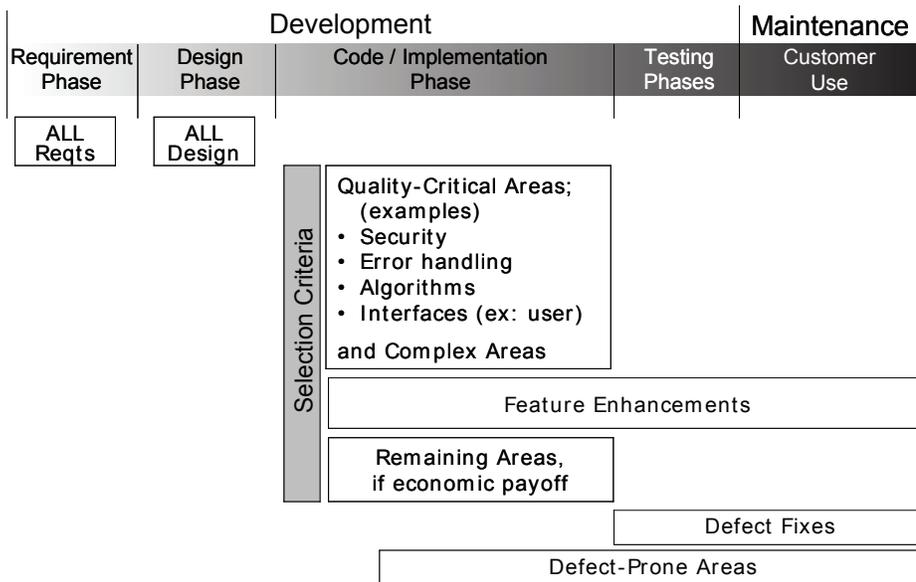


Figure 2: *Prioritizing What to Inspect*

pseudo-quality at the end of the production line.

What naturally followed was the idea of applying sampling quality control techniques to the software development life cycle as if it were a production line. Specifically, this involves sampling the product periodically (detect defects), making adjustments as defects are found (fix defects and improve the development process), and predicting the shipped product quality based on the results of the sampling.

Application of the sampling quality control techniques to the software development cycle led to the development of the software inspection process. The most widely known and practiced inspection process was introduced to the IBM software community in 1972 by a team led by Michael Fagan and managed by Lew Priven (co-author of this article) [6].

In the case of software, the development life cycle is the production line, and inspections are the sampling and prediction technique. Inspections are the vehicle to sample the product in the earlier phases of the development life cycle to detect and fix defects closest to the point of injection, and

the data collected from inspections can be used as the basis for predicting the quality of the delivered product.

How Have Inspections Evolved?

In 1972, Priven published an IBM Technical Report which described a software development management system including *points of management control* using process monitors that evolved into inspections [8]. The management system was based on a well-defined development process – which satisfied the need for a production line as described earlier. With the *production line* in place, Priven hired Michael Fagan, a quality engineer with a hardware and manufacturing background, to work with the development team to find a way to improve the quality of delivered software [6-10]. The IBM (Fagan) Inspection Process then evolved as a critical component of the end-to-end software development life cycle. Over the years, the integration of inspections into the software development life cycle has been lost as the *inspection process came to be viewed as a standalone quality process with inspection execution becoming the prime focus*. However, the supporting infrastructure of a software development life cycle is still critical to successfully imple-

menting inspections.

Why Is the Supporting Development Infrastructure Important?

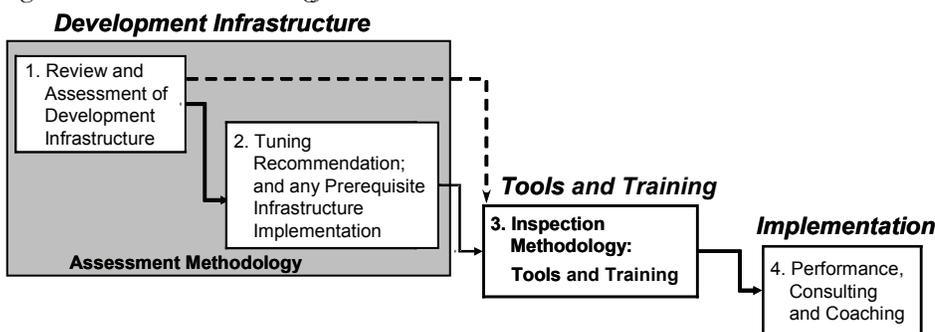
The supporting infrastructure of a well-defined development process is important because it requires management at all levels – and during all development phases – to actively support the inspection process. A life-cycle view is needed because the cost and schedule impact are primarily borne by the requirements, design, and implementation components of the organization. However, while these components also realize some of the reduced cost, higher quality, and improved schedule benefits, the majority of these benefits are primarily realized in testing and maintenance.

Theory Is Good, but Why Are Inspections Not Embraced?

In addition to being viewed as a stand-alone process, which lacks a life-cycle view of investment and associated savings, inspections have also been characterized by a number of myths. These myths discourage implementation. While there is a kernel of truth in each myth, each can be turned into a positive. Some examples follow:

- **Inspections are time consuming.** Yes, they add up-front development time to requirements, design and code. However, rather than being viewed as a problem, this additional up-front time for inspections should be viewed as an investment in obtaining the overall quality, cost, and schedule benefits over the project's life cycle.
- **Inspections are bureaucratic and one size fits all.** System engineers and software engineers, with support from management, need to have the flexibility to adjust their inspection process to the needs of the product under development. For example, the difference between inspecting software to control a jet fighter (where a defect could be a matter of life and death) and software that displays a Web form (where the impact of a defect may be an inconvenience). The former may require a broader comprehensive set of inspections while the latter could employ other visual analysis techniques to supplement a base set of inspections.
- **All work products must be inspected.** There is a lack of guidance on when, where, and how to start an inspection process. An approach to prioritizing what work products to

Figure 3: *Assessment Methodology*



inspect needs to be intelligently applied.

While these are myths that we typically hear about inspections, upon further examination they are symptoms of a much larger underlying set of issues. The remainder of the article will focus on dealing with those *issues* which we will later refer to as *inspection pitfalls*.

A Realistic Approach

Although complete inspection coverage may be ideal, a realistic approach to inspections is to formulate a set of *selection criteria* (see Figure 2). These criteria guide the identification of those areas of the product most critical to success or where problems are most likely to occur. At the least these areas should be inspected. This addresses the common complaint that there is not enough time to integrate inspections into tight schedules yet allows for using inspections for finding defects where they are most likely to cause problems.

Figure 2 addresses this no-time issue by showing the prioritization of *what to inspect* related to the development cycle phases of the project. There should be a strong focus on requirements and design, which are the

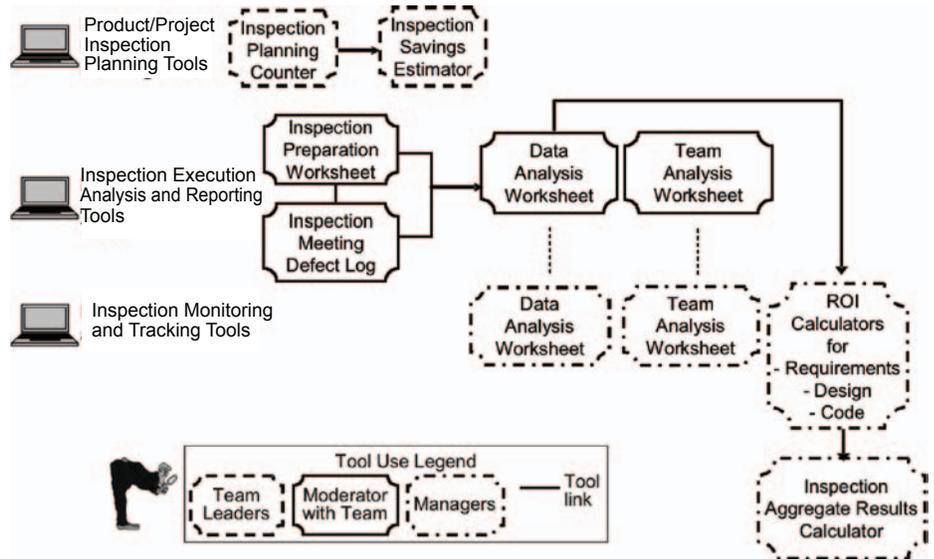


Figure 4: Computerized Inspection Tool Overview

most costly to fix when discovered later in the development cycle (see Figure 1). The focus on requirements and design is particularly important because our experience has shown that the largest numbers of defects are injected during these two phases of development. One example from a TRW study shows about 52 percent of defects are

injected in requirements and 28 percent are injected in design [11].

The most successful implementations of inspections have been in organizations that have multi-level active management support of inspections and a well-defined development life cycle with pre-existing emphasis on planning, monitoring, and

Table 1: Risks Associated With Inspection Pitfalls

No.	Pitfall	Pitfall Risks	Pitfall Solutions
1	Lack of supportive SDLC infrastructure	<ul style="list-style-type: none"> • Immature practices for planning, data collection, reporting, monitoring, and tracking • Leads to Pitfalls #3, 4, 6, 10 	<ul style="list-style-type: none"> • Assessment methodology <ul style="list-style-type: none"> ◦ Step 1: Assess client SDLC ◦ Step 2: Recommend any changes
2	Poor management understanding of the Inspection Process, its benefits, and their responsibilities	<ul style="list-style-type: none"> • Leads to Pitfalls #4, 6, 8, 9 ◦ Inadequate inspection: schedules, tools facilitation, implementation 	<ul style="list-style-type: none"> • Upper management overview • Management performance class • Student feedback from inspection class
3	No computerized management-planning tools	<ul style="list-style-type: none"> • Inadequate schedule time (Pitfall #4) • No savings appreciation, leads to no inspections or too few inspections 	<ul style="list-style-type: none"> • Planning counter tool • Savings/cost-estimator tool
4	Too little schedule time for inspections	<ul style="list-style-type: none"> • Defects escape to more costly phases to fix • Inspections not correctly executed • Leads to malicious compliance 	<ul style="list-style-type: none"> • Inspection planning counter tool • Management performance class • Criteria for prioritizing what to inspect
5	No computerized inspector tools	<ul style="list-style-type: none"> • Inconsistency, compromising shortcuts • Defects, escape to more costly phases to fix 	<ul style="list-style-type: none"> • Preparation tool • Inspection meeting tool • Data analysis tool, team analysis tool
6	Inadequate monitoring of inspection execution and tracking of results	<ul style="list-style-type: none"> • Inspection process execution deteriorates • Defects escape to more costly phases to fix • Employees lose interest when savings summaries are not periodically shared 	<ul style="list-style-type: none"> • ROI calculators for text and code • Data analysis tool, team analysis tool • Aggregate results calculator tool
7	No post-class practitioner refresher	<ul style="list-style-type: none"> • Process misunderstood, compromising shortcuts introduced, defects escape 	<ul style="list-style-type: none"> • Seminar for previous students • Inspection role-reference card • Inspection product checklist kit
8	No inspection facilitator/project champion	<ul style="list-style-type: none"> • Inspection process issues not addressed, coordinated, or resolution disseminated • Inconsistent or incorrect inspection execution • Little useful feedback to management 	<ul style="list-style-type: none"> • Upper management overview • Management performance class
9	Slow inspection implementation by project teams	<ul style="list-style-type: none"> • Ineffective start or no-start occurs • Inspection training forgotten; incorrect execution 	<ul style="list-style-type: none"> • Inspector training accommodating multiple classes, of 2 days or less, per week
10	No inspection process capture	<ul style="list-style-type: none"> • Process misunderstood, inconsistent execution, defects escape • No repository for project lessons learned 	<ul style="list-style-type: none"> • Course material tailoring • Inspection process capture tool

Road Map to Successful Inspection Implementation

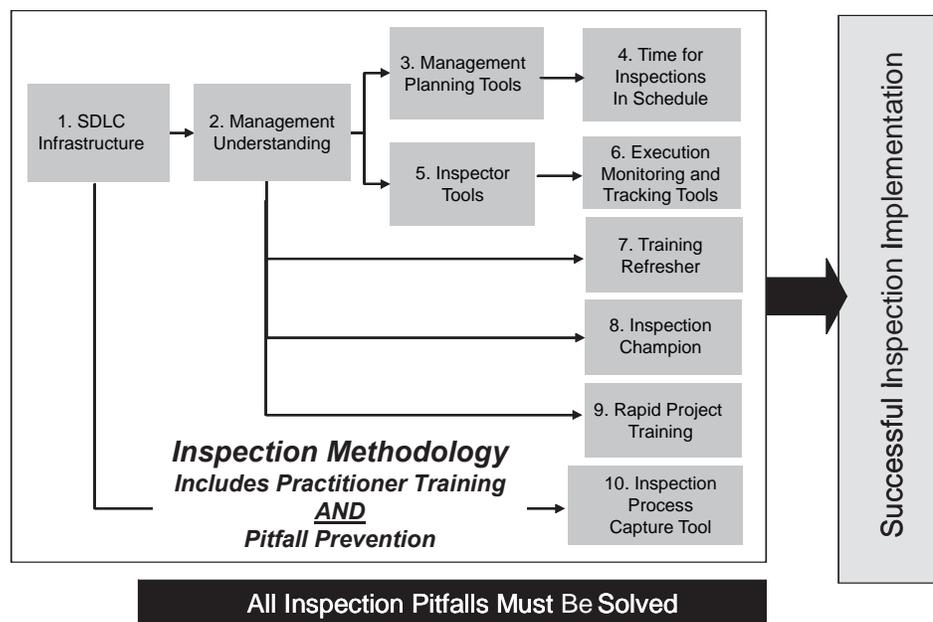


Figure 5: *Inspection Infrastructure*

measurements use.

Development Infrastructure to Support Inspections

There is a lot of guidance on the structure of inspections such as the Institute for Electronics and Electrical Engineers (IEEE) Standard 1028-1997 [12] and how to conduct an inspection, but little guidance on the following:

1. How to select what to inspect (see Figure 2).
2. How to develop an appropriate software development life-cycle infrastructure that provides the necessary framework for successful implementation of inspections.
3. How to determine what computerized tools are needed to ensure proper inspection execution and management visibility into results, project savings, and return on investment (ROI). For example, because of the lack of inspection tools, data collection – which is too often left to the discretion of the inspection teams – is often not performed or is performed inconsistently. Therefore, data needed to evaluate inspection effectiveness is not easily available to management.

These three items will be discussed further in the next section.

Successful inspection implementation requires a software development life-cycle infrastructure that demands planning, data collection, reporting, monitoring, and tracking. Introducing inspections into a

project culture that does not believe in and have a development infrastructure that actively supports these activities is fraught with risk.

Developing an appropriate infrastructure begins with selecting a framework upon which to build your development life cycle. A widely accepted framework is the Capability Maturity Model® (CMM®), and its successor CMM-Integration (CMMI®). However, as Watts Humphrey points out in [13], “Although the CMM [and CMMI] provides a powerful improvement framework, its focus is necessarily on what organizations should do – not how they should do it.”

There are four key steps to filling out the development framework:

1. Select a development model (e.g., iterative, incremental, waterfall, spiral, agile).
2. Clearly define the development life cycle by identifying and recording for each process within the life cycle, its required inputs, the input’s entrance criteria, the *what and how* of the process, the expected outputs, and the output’s exit criteria.
3. Get process agreement by all components of the development organization (e.g., requirements generators, designers, coding/implementers, testers, etc.).
4. Determine which project tools will be used for planning, data collection, reporting, monitoring, and tracking (tool examples are critical path, earned value, etc.).

When these steps are completed, the introduction of inspections has the neces-

sary framework (i.e., development infrastructure) for ongoing success and for inspections to be accepted as a very integral part of the end-to-end development life-cycle.

How an Inspection Methodology Can Reinvent Inspections

Inspections will only be successful long term if they are integral to a well-defined development process that has active management support in each phase of development. The methodology shown in Figure 3 (see page 25) starts with an assessment to ensure an adequate development life-cycle infrastructure is in place prior to inspection training. Steps 1 and 2 in Figure 3 are the assessment steps.

Once the development infrastructure is in place, what else needs to be done? Based on our experience in training more than 5,000 inspectors in companies at more than 50 locations, evaluating the data collected and observing the ongoing implementation or lack thereof, we have identified 10 inspection pitfalls, each of which inhibits inspection implementation. These inspection pitfalls must be resolved to achieve lasting benefits from inspections.

The 10 inspection pitfalls and associated risks are shown in Table 1 (see page 25). Note: The lack of a well-defined SDLC infrastructure, discussed earlier, is the first pitfall.

Table 1 identifies how each inspection pitfall leads to findable defects not being discovered with inspections, resulting in the following:

- A. Development cost savings are not fully realized.
- B. Quality improvements are not fully achieved.
- C. Maintenance and support savings are not realized.
- D. Inspections could become a total cost, not a savings.

We distinguish between the development life-cycle infrastructure – within which inspections fit (previous section), and the inspection infrastructure – which enables proper inspection execution for achieving the maximum benefit. An enabling inspection infrastructure must address all 10 pitfalls and would consist of the following:

1. Computerized management tools for use in planning inspections and predicting the overall project costs and savings from applying inspections (see Figure 4 on page 25 for an inspection tool overview example).
2. Computerized tools to aid inspectors

* CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

- in correctly and consistently performing inspections, gathering inspection related data, and performing analysis to identify how future inspections can be improved.
- Monitoring and analysis computerized tools for management's post-inspection evaluation of individual inspections.
 - Computerized management tools for analyzing inspection ROI and an aggregate calculator for assessing and tracking the resulting project savings from multiple inspections.
 - An inspection process that provides flexibility for prioritizing what to inspect as shown in Figure 2.
 - Ability to have customized training material which incorporates your terminology and is based on your needs.
 - Rapid training (two days or less) of project personnel in a comprehensive training course with significant focus on requirements and design.
 - An overview briefing for upper management along with a more rigorous management performance course so upper managers and project leaders can fully understand the inspection process, its benefits, and their responsibilities
 - Follow-up practitioner refreshers to deal with any implementation problems – focused on making inspection implementation successful both initially and long-term.
 - An inspection process capture tool to enable inspections to quickly become an integral part of a company's SDLC infrastructure.

Table 1 shows how an *Inspection Methodology* can solve and prevent the 10 inspection pitfalls.

The Road Map to Success

The pitfall solution road map in Figure 5 shows the solution relationships that provide for successful software inspection implementation that will endure over the long term. The pitfall solutions provide the inspection infrastructure that, together with a comprehensive inspector training program, forms an inspection methodology for achieving a lasting and successful inspection program.

Summary

Our experience has shown us that inspections can live up to their potential and be embraced by the development community if the following happens:

- Inspections are integral to a well-defined software development life-cycle infrastructure supported by man-

agement in each phase of development.

- Inspections are flexible in determining what to inspect.
- Computerized tools are available to assist management in planning inspections and estimating project savings before commitment.
- Computerized tools are available to guide the inspection teams.
- Management tools are available for monitoring inspection process conformance (not an individual's performance) and tracking resulting inspection benefits.
- Project personnel are provided with the proper training and follow-up support. ♦

References

- NIST. "The Economic Impacts of Inadequate Infrastructure for Software Testing." NIST Planning Report 02-3. May 2002.
- Bennett, Ted L., and Paul W. Wennberg. "Eliminating Embedded Software Defects Prior to Integration Test." CROSSTALK Dec. 2005.
- McConnell, Steve. "Best Influences on Software Engineering Over Past 50 Years." *IEEE Software* Jan./Feb. 2000.
- Wieggers, Karl. "The More Things Change." *Better Software* Oct. 2006.
- Jones, Capers. "Interview." *Computer Aid Inc.* July 2005.
- Radice, Ron. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Weller, Ed. "Calculating the Economics of Inspections." *StickyMinds* Jan. 2002.
- Priven, L.D. "Managing the Programming Development Cycle." IBM Technical Report 21.463 Mar.1972.
- Fagan, Michael E. "Design and Code Inspections and Process Control in the Development of Programs." *IBM Technical Report 21.572*. Dec. 1974.
- Priven, L. and F. Tsui. "Implementation of Quality Control in Software Development." *AFIPS Conference Proceedings, 1976 National Computer Conference* 45 (1976):443-449.
- McGraw, Gary. *Making Essential Software Work*. Cigital, Inc. Mar. 2003 <<http://cigital.com/whitepapers>>.
- Software Engineering Standards Committee of the IEEE Computer Society. "IEEE Standard for Software Reviews, Section 6. Inspections." IEEE Std. 1028-1997, 1997.
- Humphrey, Watts. "Three Dimensions of Process Maturity." CROSSTALK Feb. 1998.

About the Authors



Roger Stewart is co-founder of the Stewart-Priven Group. Previously, he spent 30 years with IBM's Federal Systems Division managing and developing systems for air traffic control, satellite command and control, on-board space shuttle, LAMPS helicopter and in commercial banking, telecommunication and networking systems. Stewart has a bachelor's degree in mathematics from Cortland University.

The Stewart-Priven Group

7962 Old Georgetown RD

STE B

Bethesda, MD 20814

Phone: (865) 458-6685

Fax: (865) 458-9139

E-mail: spgroup@charter.net



Lew Priven is co-founder of the Stewart-Priven Group. He is an experienced executive with systems and software management and technical background. Priven was vice-president of engineering and application development at General Electric Information Services and vice president of application development for IBM's Application Systems Division. He has a bachelor's degree in electrical engineering from Tufts University and a master's degree in management from Rensselaer Polytechnic Institute.

The Stewart-Priven Group

7962 Old Georgetown RD

STE B

Bethesda MD 20814

Phone: (865) 458-6685

Fax: (865) 458-9139

E-mail: spgroup@charter.net



Computer Science Education: Where Are the Software Engineers of Tomorrow?

By Dr. Robert B.K. Dewar and Dr. Edmond Schonberg
AdaCore Inc.

It is our view that Computer Science (CS) education is neglecting basic skills, in particular in the areas of programming and formal methods. We consider that the general adoption of Java as a first programming language is in part responsible for this decline. We examine briefly the set of programming skills that should be part of every software professional's repertoire.

It is all about programming! Over the last few years we have noticed worrisome trends in CS education. The following represents a summary of those trends:

1. Mathematics requirements in CS programs are shrinking.
2. The development of programming skills in several languages is giving way to cookbook approaches using large libraries and special-purpose packages.
3. The resulting set of skills is insufficient for today's software industry (in particular for safety and security purposes) and, unfortunately, matches well what the outsourcing industry can offer. We are training easily replaceable professionals.

These trends are visible in the latest curriculum recommendations from the Association for Computing Machinery (ACM). Curriculum 2005 does not mention mathematical prerequisites at all, and it mentions only one course in the theory of programming languages [1].

We have seen these developments from both sides: As faculty members at New York University for decades, we have regretted the introduction of Java as a first language of instruction for most computer science majors. We have seen how this choice has weakened the formation of our students, as reflected in their performance in systems and architecture courses. As founders of a company that specializes in Ada programming tools for mission-critical systems, we find it harder to recruit qualified applicants who have the right foundational skills. We want to advocate a more rigorous formation, in which formal methods are introduced early on, and programming languages play a central role in CS education.

Formal Methods and Software Construction

Formal techniques for proving the correctness of programs were an extremely active subject of research 20 years ago. However,

the methods (and the hardware) of the time prevented these techniques from becoming widespread, and as a result they are more or less ignored by most CS programs. This is unfortunate because the techniques have evolved to the point that they can be used in large-scale systems and can contribute substantially to the reliability of these systems. A case in point is the use of SPARK in the re-engineering of the ground-based air traffic control system in the United Kingdom (see a description of iFACTS – Interim Future Area Control Tools Support, at <www.nats.co.uk/article/90>). SPARK is a subset of Ada augmented with assertions that allow the designer to prove important properties of a program: termination, absence of runtime exceptions, finite memory usage, etc. [2]. It is obvious that this kind of design and analysis methodology (dubbed Correctness by Construction) will add substantially to the reliability of a system whose design has involved SPARK from the beginning. However, PRAXIS, the company that developed SPARK and which is designing iFACTS, finds it hard to recruit people with the required mathematical competence (and this is present even in the United Kingdom, where formal methods are more widely taught and used than in the United States).

Another formal approach to which CS students need exposure is model checking and linear temporal logic for the design of concurrent systems. For a modern discussion of the topic, which is central to mission-critical software, see [3].

Another area of computer science which we find neglected is the study of floating-point computations. At New York University, a course in numerical methods and floating-point computing used to be required, but this requirement was dropped many years ago, and now very few students take this course. The topic is vital to all scientific and engineering software and is semantically delicate. One would imagine that it would be a required part of all courses in scientific computing, but these often

take MatLab to be the universal programming tool and ignore the topic altogether.

The Pitfalls of Java as a First Programming Language

Because of its popularity in the context of Web applications and the ease with which beginners can produce graphical programs, Java has become the most widely used language in introductory programming courses. We consider this to be a misguided attempt to make programming more fun, perhaps in reaction to the drop in CS enrollments that followed the dot-com bust. What we observed at New York University is that the Java programming courses did not prepare our students for the first course in systems, much less for more advanced ones. Students found it hard to write programs that did not have a graphic interface, had no feeling for the relationship between the source program and what the hardware would actually do, and (most damaging) did not understand the semantics of pointers at all, which made the use of C in systems programming very challenging.

Let us propose the following principle: The irresistible beauty of programming consists in the reduction of complex formal processes to a very small set of primitive operations. Java, instead of exposing this beauty, encourages the programmer to approach problem-solving like a plumber in a hardware store: by rummaging through a multitude of drawers (i.e. packages) we will end up finding some gadget (i.e. class) that does roughly what we want. How it does it is not interesting! The result is a student who knows how to put a simple program together, but does not know how to program. A further pitfall of the early use of Java libraries and frameworks is that it is impossible for the student to develop a sense of the run-time cost of what is written because it is extremely hard to know what any method call will eventually execute. A lucid analysis of the problem is presented in [4].

We are seeing some backlash to this

approach. For example, Bjarne Stroustrup reports from Texas A & M University that the industry is showing increasing unhappiness with the results of this approach. Specifically, he notes the following:

I have had a lot of complaints about that [the use of Java as a first programming language] from industry, specifically from AT&T, IBM, Intel, Bloomberg, NI, Microsoft, Lockheed-Martin, and more. [5]

He noted in a private discussion on this topic, reporting the following:

It [Texas A&M] did [teach Java as the first language]. Then I started teaching C++ to the electrical engineers and when the EE students started to out-program the CS students, the CS department switched to C++. [5]

It will be interesting to see how many departments follow this trend. At AdaCore, we are certainly aware of many universities that have adopted Ada as a first language because of similar concerns.

A Real Programmer Can Write in Any Language (C, Java, Lisp, Ada)

Software professionals of a certain age will remember the slogan of old-timers from two generations ago when structured programming became the rage: Real programmers can write Fortran in any language. The slogan is a reminder of how thinking habits of programmers are influenced by the first language they learn and how hard it is to shake these habits if you do all your programming in a single language. Conversely, we want to say that a competent programmer is comfortable with a number of different languages and that the programmer must be able to use the mental tools favored by one of them, even when programming in another. For example, the user of an imperative language such as Ada or C++ must be able to write in a functional style, acquired through practice with Lisp and ML¹, when manipulating recursive structures. This is one indication of the importance of learning in-depth a number of different programming languages. What follows summarizes what we think are the critical contributions that well-established languages make to the mental tool-set of real programmers. For example, a real programmer should be able to program inheritance and dynamic dispatching in C, information hiding in Lisp,

tree manipulation libraries in Ada, and garbage collection in anything but Java. The study of a wide variety of languages is, thus, indispensable to the well-rounded programmer.

Why C Matters

C is the low-level language that everyone must know. It can be seen as a portable assembly language, and as such it exposes the underlying machine and forces the student to understand clearly the relationship between software and hardware. Performance analysis is more straightforward, because the cost of every software statement is clear. Finally, compilers (GCC for example) make it easy to examine the generated assembly code, which is an excellent tool for understanding machine language and architecture.

Why C++ Matters

C++ brings to C the fundamental concepts of modern software engineering: encapsulation with classes and namespaces, information hiding through protected and private data and operations, programming by extension through virtual methods and derived classes, etc. C++ also pushes storage management as far as it can go without full-blown garbage collection, with constructors and destructors.

Why Lisp Matters

Every programmer must be comfortable with functional programming and with the important notion of referential transparency. Even though most programmers find imperative programming more intuitive, they must recognize that in many contexts that a functional, stateless style is clear, natural, easy to understand, and efficient to boot.

An additional benefit of the practice of Lisp is that the program is written in what amounts to abstract syntax, namely the internal representation that most compilers use between parsing and code generation. Knowing Lisp is thus an excellent preparation for any software work that involves language processing.

Finally, Lisp (at least in its lean Scheme incarnation) is amenable to a very compact self-definition. Seeing a complete Lisp interpreter written in Lisp is an intellectual revelation that all computer scientists should experience.

Why Java Matters

Despite our comments on Java as a first or only language, we think that Java has an important role to play in CS instruction. We will mention only two aspects of the language that must be part of the real programmer's skill set:

1. An understanding of concurrent programming (for which threads provide a basic low-level model).
2. Reflection, namely the understanding that a program can be instrumented to examine its own state and to determine its own behavior in a dynamically changing environment.

Why Ada Matters

Ada is the language of software engineering par excellence. Even when it is not the language of instruction in programming courses, it is the language chosen to teach courses in software engineering. This is because the notions of strong typing, encapsulation, information hiding, concurrency, generic programming, inheritance, and so on, are embodied in specific features of the language. From our experience and that of our customers, we can say that a real programmer writes Ada in any language. For example, an Ada programmer accustomed to Ada's package model, which strongly separates specification from implementation, will tend to write C in a style where well-commented header files act in somewhat the same way as package specs in Ada. The programmer will include bounds checking and consistency checks when passing mutable structures between subprograms to mimic the strong-typing checks that Ada mandates [6]. She will organize concurrent programs into tasks and protected objects, with well-defined synchronization and communication mechanisms.

The concurrency features of Ada are particularly important in our age of multi-core architectures. We find it surprising that these architectures should be presented as a novel challenge to software design when Ada had well-designed mechanisms for writing safe, concurrent software 30 years ago.

Programming Languages Are Not the Whole Story

A well-rounded CS curriculum will include an advanced course in programming languages that covers a wide variety of languages, chosen to broaden the understanding of the programming process, rather than to build a résumé in perceived hot languages. We are somewhat dismayed to see the popularity of scripting languages in introductory programming courses. Such languages (Javascript, PHP, Atlas) are indeed popular tools of today for Web applications. Such languages have all the pedagogical defaults that we ascribe to Java and provide no opportunity to learn algorithms and performance analysis. Their absence of strong typing leads to a trial-



Get Your Free Subscription

Fill out and send us this form.

517 SMXS/MXDEA

6022 FIR AVE

BLDG 1238

HILL AFB, UT 84056-5820

FAX: (801) 777-8069 DSN: 777-8069

PHONE: (801) 775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

NAME: _____

RANK/GRADE: _____

POSITION/TITLE: _____

ORGANIZATION: _____

ADDRESS: _____

BASE/CITY: _____

STATE: _____ ZIP: _____

PHONE: (____) _____

FAX: (____) _____

E-MAIL: _____

CHECK BOX(ES) TO REQUEST BACK ISSUES:

OCT2006 STAR WARS TO STAR TREK

NOV2006 MANAGEMENT BASICS

DEC2006 REQUIREMENTS ENG.

JAN2007 PUBLISHER'S CHOICE

FEB2007 CMMI

MAR2007 SOFTWARE SECURITY

APR2007 AGILE DEVELOPMENT

MAY2007 SOFTWARE ACQUISITION

JUNE2007 COTS INTEGRATION

JULY2007 NET-CENTRICITY

AUG2007 STORIES OF CHANGE

SEPT2007 SERVICE-ORIENTED ARCH.

OCT2007 SYSTEMS ENGINEERING

NOV2007 WORKING AS A TEAM

DEC2007 SOFTWARE SUSTAINMENT

TO REQUEST BACK ISSUES ON TOPICS NOT LISTED ABOVE, PLEASE CONTACT <STSC.CUSTOMERSERVICE@HILL.AF.MIL>.

and-error programming style and prevents students from acquiring the discipline of separating design of interfaces from specifications.

However, teaching the right languages alone is not enough. Students need to be exposed to the tools to construct large-scale reliable programs, as we discussed at the start of this article. Topics of relevance are studying formal specification methods and formal proof methodologies, as well as gaining an understanding of how high-reliability code is certified in the real world. When you step into a plane, you are putting your life in the hands of software which had better be totally reliable. As a computer scientist, you should have some knowledge of how this level of reliability is achieved. In this day and age, the fear of terrorist cyber attacks have given a new urgency to the building of software that is not only bug free, but is also immune from malicious attack. Such high-security software relies even more extensively on formal methodologies, and our students need to be prepared for this new world. ♦

References

1. Joint Taskforce for Computing Curricula. "Computing Curricula 2005: The Overview Report." ACM/AIS/IEEE, 2005 <www.acm.org/education

/curric_vols/CC2005-March06Final.pdf>.

2. Barnes, John. High Integrity Ada: The Spark Approach. Addison-Wesley, 2003.
3. Ben-Ari, M. Principles of Concurrent and Distributed Programming. 2nd ed. Addison-Wesley, 2006.
4. Mitchell, Nick, Gary Sevitsky, and Harini Srinivasan. "The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications." Workshop on Library-Centric Software Design, Object-Oriented Programming, Systems, Languages, and Applications, San Diego, CA, 2005.
5. Stroustrup, Bjarne. Private communication. Aug. 2007.
6. Holzmann Gerard J. "The Power of Ten – Rules for Developing Safety Critical Code." IEEE Computer June 2006: 93-95.

Note

1. Several programming language and system names have evolved from acronyms whose formal spellings are no longer considered applicable to the current names for which they are readily known. ML, Lisp, GCC, PHP, and SPARK fall under this category.

About the Authors



Robert B.K. Dewar, Ph.D., is president of AdaCore and a professor emeritus of computer science at New York University. He has been involved in the design and implementation of Ada since 1980 as a distinguished reviewer, a member of the Ada Rapporteur group, and the chief architect of Gnu Ada Translator. He was a member of the Algol68 committee and is the designer and implementor of Spitbol. Dewar lectures widely on programming languages, software methodologies, safety and security, and on intellectual property rights. He has a doctorate in chemistry from the University of Chicago.

AdaCore
104 Fifth AVE
15th FL
New York, NY 10011
Phone: (212) 620-7300 ext. 100
Fax: (212) 807-0162
E-mail: dewar@adacore.com



Edmond Schonberg, Ph.D., is vice-president of AdaCore and a professor emeritus of computer science at New York University. He has been involved in the implementation of Ada since 1981. With Robert Dewar and other collaborators, he created the first validated implementation of Ada83, the first prototype compiler for Ada9X, and the first full implementation of Ada2005. Schonberg has a doctorate in physics from the University of Chicago.

AdaCore
104 Fifth AVE
15th FL
New York, NY 10011
E-mail: schonberg@adacore.com



Bite My Bytes

I've been doing some networking lately. Not the kind of networking that gets one a new job, but computer networking. Since job hunting networking is now done on a computer, some folks may not see the difference, so just take my word for it, networking isn't necessarily networking. I've been doing the type of networking that allows computers to talk to each other. This is what some people would call network administration or network engineering.

There are many different types of people in the networking field, with various backgrounds. Some of them don't have technical backgrounds, and technical fields have their own language, or vocabulary.

While working on an infrared project some years ago, I discovered that there existed three different definitions for infrared. One definition exists for engineers, which was defined in the Institute for Electronics and Electrical Engineers (IEEE) standard, one for physics, and one for astronomers. The different definitions evolved separately from a historical perspective, and that is understandable. Astronomers have been around long before engineers.

Now, in the networking literature, I find a particular word, the byte, that seems to be taking on different meanings. As an engineer, I learned that a byte is eight bits. Always. A nibble is four bits. A word is the size of the processor data line, or the number of data bits that the processor takes in on a clock cycle. Sometimes this is the size of the internal data bus, but not always. When I talk about a byte, I mean eight bits. When someone else talks about a byte, I think they mean eight bits. As the March Hare told Alice while she was visiting Wonderland, "Then you should say what you mean [1]." We have to use words with denotations that are shared.

In several books that I have been reading on computer networking, the authors don't seem to know that a byte is eight bits – always. Let's review some computer science. A nibble is four bits, a byte is eight bits, and a computer word varies, depending on the computer architecture. Some early computers were six bit machines, but most of us are familiar with the eight bit, 16 bit, 32 bit, and now 64 bit machines. I think that some of these networking authors are confusing bytes with computer words. Could that be because of different backgrounds making up the network engineering field? Maybe we will have different definitions of a byte, one from the engineering field, one from the computer scientists, and one from certified network engineers. Is it too late to stop this madness?

One book gives the correct definition for byte on page 83, then, on page 87, defines a byte as seven or eight bits, depending on whether parity is used [2]. This book, which is very good, has a glossary where a byte is correctly defined as eight bits. Perhaps, on page 87, the author is confusing using bytes with using the American Standard Code for Information Interchange (ASCII) character set. The ASCII is a seven-bit code, or eight, if parity is used. A byte is always eight bits.

So, what does dictionary.com say? To my horror, some of the definitions given online differ. But my Webster's has it right [3]. It defines a byte as "a group of eight binary digits processed as a unit by a computer and used especially to represent an alphanumeric character." And Webster's definition for word, under 2c, has "a number of bytes processed as a unit and conveying a quantum of information in communication and computer work." Webster's does not have a technical definition of a nibble.

Again, to my horror, I found that the IEEE Standard Glossary

of Software Engineering Terminology did not commit a byte to always be eight bits. The definition given for byte and the one given for word, was very similar, and a definition for nibble was not given at all [4].

I think that because a byte is often used to represent an alphanumeric character, and the ASCII code is 7 or 8 bits, depending on whether parity is used, some confusion is creeping in here. ASCII is being replaced by Unicode, which can use as many as four bytes (or 32 bits). See <<http://www.unicode.org>>. Now, if four bytes are 32 bits, how large is a byte?

Some of these networking book authors don't think that a byte is always eight bits, and they want to use the word octet to convey an eight bit quantity [5]. I was just wondering if the definition of byte was being changed because of differing backgrounds of people in the information technology workplace. Before reading these books, it never occurred to me that anyone thought that a byte was anything other than eight bits.

In Salinger's book, "The Catcher in the Rye," Holden Caulfield tells his little sister Phoebe what he would like to do with his life [6]. He tells of picturing these children playing in a big field of rye next to a cliff. He wants to stand on the edge of the cliff and catch any kids who fall over. In my version of this fantasy, I see information technology professionals wandering about in a big field of rye wondering if a byte had six, seven, or eight bits in it. I would be there to assure them that every byte has eight bits and only eight bits. It's the size of a computer word that varies.

— Dennis Ludwig
dennis.ludwig@wpafb.af.mil

References

1. Carroll, Lewis. Alice in Wonderland. Grosset and Dunlap Publishers, 1980.
2. Lammle, Todd. CCNA: Cisco Certified Network Associate. Wiley Publishing, Inc., 2005.
3. Merriam-Wester's Collegiate Dictionary, 10th ed. (2001).
4. IEEE. IEEE Std. 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology.
5. Comer, Douglas E. Internetworking with TCP/IP: Principles, Protocols, and Architecture. Pearson Prentice Hall, 2006.
6. Salinger, J.D. The Catcher in the Rye. Little, Brown, and Company, 2001.

Can You BACKTALK?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CROSSTALK, we also accept articles for the BACKTALK column. BACKTALK articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BACKTALK article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

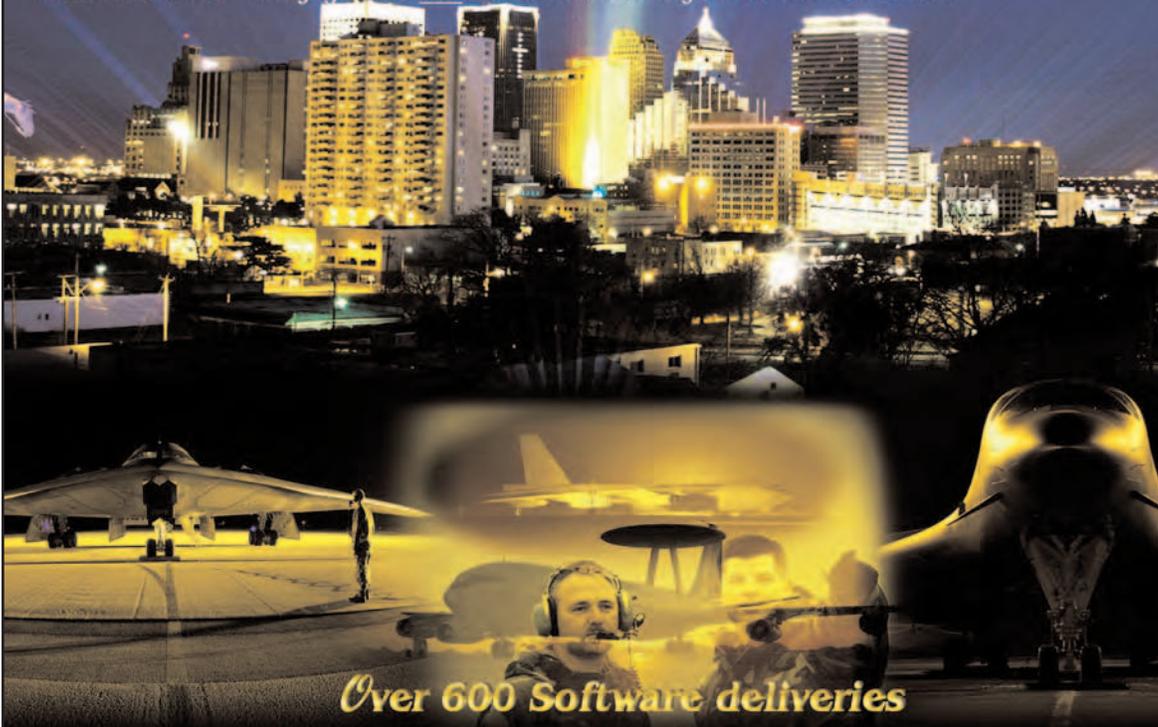
For a complete author's packet detailing how to submit your BACKTALK article, visit our Web site at <www.stsc.hill.af.mil>.



Software Engineering

As leaders in the avionics software industry, the Software Professionals at OC-ALC have a proven track record of producing software On-Time, On-Budget, and Defect Free.

Our staff of engineering professionals, as well as our industry partners, are committed to providing our customers with software and engineering solutions that make our Warfighters the most dominant forces in the world.



**Over 600 Software deliveries
FY 04-07 99.5% On-Time
Can you find a better S/W Supplier?**

- Weapon System Software
- Automated Test Equipment Software
- Engine Test Cells
- Industrial Automation
- Application Development

Oklahoma City Air Logistics Center
76th Software Maintenance Group
Phil Perkins (Deputy Director)
(405) 736.3341
DSN: 336.3341
phil.perkins@tinker.af.mil

CROSSTALK / 517 SMXS/MXDEA

6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSR STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CROSSTALK is co-sponsored by the following organizations:



NAV  AIR



Homeland Security