Future Directions in Software Engineering
How long can you wait for CMMI® Compliance?

Manage your software and systems development in guaranteed compliance with the CMMI NOW!

processMax includes all the necessary policies, procedures, guidelines, criteria, templates, and forms with role-based, step-by-step instructions, ready for use—everything you need for CMMI compliance. Integrated with robust document management and workflow, processMax is the intranet web-based solution for effective and efficient management of your development and maintenance projects. processMax also includes a complete measurement and reporting capability. Measurement collection and report generation are fully automated—giving managers real-time, fact-based insight into all aspects of project progress, including earned value. processMax is the proven solution—as verified by more than 60 successful independent appraisals.

pragma SYSTEMS CORPORATION
www.pragmasystems.com
877.838.PMAX
info@pragmasystems.com

GSA Schedule Contract No. GS-35F-0559S. processMax is a registered trademark of pragma SYSTEMS CORPORATION.
CMMI is a registered trademark of Carnegie Mellon University. Copyright © 2007 pragma SYSTEMS CORPORATION.
Since the emergence of SW engineering in the 1960s, the size, pervasiveness, and complexity of software and software intensive systems have increased by several orders of magnitude. The Nidiffer article illustrates this by noting that the size of aircraft software systems in the 1960s approximated 1,000 lines of code while aircraft systems built in 2000 contained more than six million lines of code. Additionally, both Boehm and Nidiffer cite the graph shown in Figure 1 which illustrates that the pervasiveness of software within aircraft systems has increased from less than 10% in the 1960s to 80% in 2000.

We know that increases in software size reflect, as well as contribute to, increased complexity, which, in turn, has contributed to pushing delivery and costs well beyond targeted schedules and budgets.

In 2006, in response to a request from its OSD sponsor, the National Defense Industrial Association (NDIA) conducted a workshop to identify the top software issues relative to the acquisition and deployment of Software-Intensive-Systems (SIS). The subsequent workshop report identified seven issues and some recommendations for addressing those issues (see Figure 2).[1] Then in March, 2007, OUSD hosted a software workshop with the purpose of reviewing ongoing initiatives and identifying gaps — software focus areas not currently supported by an initiative or associated activities — as an initial step in moving forward with the NDIA recommendations. The resulting identified gaps are listed in Figure 2.

Note that these top issues are addressed in significant detail in Dr. Nidiffer’s article. Most of these issues (and gaps) have existed for a long time but that does not mean that the Software Engineering community has not been making progress. It does, however, communicate the complexity of the issues themselves and perhaps the possibility that they will always be issues presenting difficult challenges because of the rapid pace of technology within our field. We may never get to that comfort zone where we can relax and say, “Well, we have software testing under control --- that’s no longer an issue”, or “we know how to sustain systems now, even System-of-Systems --- no more lessons to be learned there”. It is also noteworthy

---

[1] Note: This graph was provided by authors, Dr. Barry Boehm and Dr. Kenneth Nidiffer and is referenced in their articles. The basic graph originated at the Defense Systems Management College (DSMC); overlayed data and title were provided by author, Dr. Kenneth Nidiffer.
Challenges Dominate Our Future

Continued from page 3.

<table>
<thead>
<tr>
<th>NDIA Top SW Issues and Recommendations - 2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The impact of system requirements upon software is not consistently quantified and managed in development or sustainment.</td>
</tr>
<tr>
<td>2. Fundamental system engineering decisions are made without full participation of software engineering.</td>
</tr>
<tr>
<td>3. Software life-cycle planning and management by acquirers and suppliers is ineffective.</td>
</tr>
<tr>
<td>4. The quantity and quality of software engineering expertise is insufficient to meet the demands of government and the defense industry.</td>
</tr>
<tr>
<td>5. Traditional software verification techniques are costly and ineffective for dealing with the scale and complexity of modern systems.</td>
</tr>
<tr>
<td>6. There is a failure to assure correct, predictable, safe, secure execution of complex software in distributed environments.</td>
</tr>
<tr>
<td>7. Inadequate attention is given to total lifecycle issues for COTS/NDI impacts on lifecycle cost and risk.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GAPS Identified March, 2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>(No activity/ownership)</td>
</tr>
</tbody>
</table>

that most of these issues relate to planning, management, and software engineering expertise, reflecting the human aspects of software engineering. Perhaps the triangle that is often used to represent software engineering as an equal balance of people, process, and technology, needs to be shifted to place a greater emphasis on the “people” vertex in the future.

This issue of Software Tech News (STN) contains articles that look at the future of Software Engineering from a variety of perspectives.

In the first article, Dr. Barry Boehm, provides a high-level view of the future of SW Engineering; he presents a collection of challenges and offers some general guidance for what we must do to address those challenges. He concludes that software will play a significant role in determining how well we all get along in this global community, a highly thought provoking prediction.

In the 2nd article, Dr. Ken Nidiffer, who currently directs strategic plans for government programs at the Software Engineering Institute (SEI), takes a look at our software engineering history since the 1960s and cautions about not losing sight of the lessons we have already learned, so that we don’t repeat our mistakes. Then, he focuses on the specific software issues identified in the NDIA report [1] and provides details regarding how DoD plans to address them. He identifies, among other things, human-factor/human-interaction technologies
Challenges Dominate Our Future

Continued from page 4.

as becoming more dominate in the field. He implies that as our capabilities grow, so does our vision of what we want/need, so there may always be a question as to our ability to achieve DoD’s goal of deterrence and dominance based on information superiority. He observes that, historically, software engineering has progressed through a series of plateaus or bottlenecks where it was difficult to develop the next level of complex systems. At these points, thought leaders emerged and introduced different ways of thinking, which allowed us to move forward. Dr. Nidiffer senses we have reached another plateau with the need to develop net-centric systems, federated systems and system of systems. Ideas exist but adequate resources may not be available to research and mature these new concepts that will allow the software engineering community to go above this plateau, given the magnitude of problems that need to be addressed. He speaks to the inherent uncertainty in deciding our direction for the future.

Next, we present the NAVY’s near term tactical approach to dealing with some of these identified challenges. The article by Mr. Chris Miller, Program Executive Officer, Command, Control, Communications, Computers and Intelligence (PEO C4I), describes the NAVY’s C4I Open Architecture Strategy. Miller characterizes a huge disparity between the general capabilities provided by current technology and the reality of what is actually available to the sailor warfighter. I quote him, “deployed bandwidth per sailor is less than that of a typical cell phone.” He outlines PEO C4I’s aggressive plan to change both their technical model and business approach, embodied in the NAVY Open Architecture (OA) strategy, to deliver increased warfighting capabilities in a shorter time at reduced cost. His description of the reality of migrating to the NAVY OA illustrates the immense complexity of the challenges that must overcome.

The last article provides a pragmatic approach to the issue of software testing in the complex environments where software resides today and in the future. The authors, Dustin and Gauf, who are experts in software testing practices, present a business case for Automated Software Testing (AST), describing the benefits in terms of accuracy and cost, and describing how and when to plan for AST to avoid premature failure. Results of a recent survey of the software testing community, conducted by IDT, indicate that while 73% of respondents believe AST is beneficial, they cite lack of time and lack of expertise as the reason for not doing it, even though the major benefit of ATS is significant reduction is testing time. Thus, the dominate challenge for the future is to deal with the estimation, planning, and management issues that will support a migration from a somewhat reactionary, after-the-fact testing approach to a proactive approach that builds automated testing into the software life-cycle. We must walk before we can run. We cannot hope to reap the benefits of model driven testing until we get better at automating the testing process.

References

About the Author
Ellen Walker, DACS Deputy Director and Managing Editor of the DACS Software Tech News, is currently developing a series of publications on software “best practices” as part of the DACS Gold Practice Initiative. She has more than 20 years experience as a software developer in various roles spanning the entire software life cycle including project management of multiple business process re-engineering efforts within the DoD community. She is also experienced with assessment initiatives such as the Capability Maturity Model for Software (CMM-SW) and the quality management practices of the New York State Quality Award program. Ellen has an MS in Management Science (State University of New York (SUNY) at Binghamton), and bachelor degrees in both Computer Science (SUNY – Utica/Rome) and Mathematics (LeMoyne College).

Author Contact Information
Phone: 315-334-4936
Email: ewalker@dacs.dtic.mil  ellen.walker@itt.com
In looking at 40 years of aircraft requirements, there is a clear trend in software pervasiveness. The percent of software requirements involving software control has increased from less than 10% in the 1960s to 80% in the year 2000. (Note, a detailed graph of this data is provided in the Tech Views article of this issue). This makes software engineering skills increasingly valuable and software careers increasingly influential, but it also places significant responsibilities on software engineers to ensure that their software will be able to deliver high levels of dependability. Some additional future trends discussed below will make this goal increasingly challenging, but also increasingly important to address. These trends are: uncertainty and emergence; rapid change; multifaceted dependability; diversity; and interdependence.

Uncertainty and Emergence

In the past, a sequential, requirements-first process was a feasible approach for software development, because the requirements for most systems could be fairly accurately specified in advance, and used to drive decisions about design and development solutions. Increasingly, though, the nature of both the system's requirements and preferred solution elements are becoming much less feasible to specify in advance, due to such factors as the emergence of new technologies (global positioning satellites or the world wide web), changes in consumer demand patterns, changes in leadership personnel with differing priorities, or changes in desired user interface characteristics that are not prespecifiable but emerge with use. Frequently, when users are asked to specify in advance how they would like to interact with a new application, they will provide an IKIWISI (I'll Know It When I See It) response.

In such cases, using a sequential, requirements-specification-first waterfall model will generally be a recipe for a nonresponsive system and a great deal of expensive rework. It will be better to accept that your project will start at the left of a Cone of Uncertainty, as shown in Figure 1. This figure was derived from several years of experience in preparing cost estimates at various stages of system definition at TRW [Boehm, 1981]. It was dubbed the Cone of Uncertainty by Steve McConnell in [McConnell, 1998].

The best way to narrow a Cone of Uncertainty is to buy information to avoid risk. This involves identifying candidate investments in reducing uncertainty such as prototyping, simulating, modeling, benchmarking, reference checking, COTS evaluation, or market trend analysis; analyzing their relative

![Figure 1. The Cone of Uncertainty in Software Cost and Size Estimation](image-url)
Future Challenges and Rewards

Continued from page 6.

costs and benefits, and choosing the most cost-effective ways to reduce your uncertainty and risk.

However, it will also be important to organize future projects with enough agility to be able to adapt to the additional emergence of new sources of unpredictable change. As I’ll discuss further below, this involves building agility and adaptability into your projects’ staffing, organization, product architectures, and process architectures.

Rapid Change

Gone are the days when people could spend their entire careers doing the same thing in the same way. A strong case can be made that the most significant challenges facing 21st century organizations will be their ability to adapt to rapid and unpredictable change in more rapid and appropriate ways than their competitors. Evidence of this accelerated pace of change can be seen such books as The World is Flat [Friedman, 2005], showing the accelerated pace of change in consumer preferences, international competition, and business practices.

All of this change comes at a price. People enjoy the fruits of change but generally dislike having to modify their behavior. Software engineers concerned with improving their process maturity often think that Level 5 (optimizing) is where they become mature and optimize their processes for all time. This runs the risk of being overoptimized and going the way of the dinosaurs.

There will also be a lot of tensions between people and organizations rapidly adapting to change and those who prefer not to. A good example nowadays is the interaction between software developers trying to be adaptive to change within legacy fixed-price, build-to-specification software contract structures. In the commercial world, better contact structures such as “shared destiny” models are being developed; getting these refined and more generally adopted will be important to software success. Another challenge is to determine how best to integrate current plan-driven methods with emerging agile methods [Highsmith, 2002; Boehm-Turner, 2004].

A further challenge in dealing with rapid change will be to separate obsolete practices from enduring principles that need to be conserved. Some other implications for software engineers’ careers are that learning how to learn will be more important than learning things, and that some goals, such as trying to determine “the” best architecture for a system, will be better thought of as concurrently determining architecture baselines along with architecture evolution processes. The evolution of the Arpanet architecture into the architecture of the Internet is a good example.

Multifaceted Dependability

But along with improving agility, future projects will need to improve the dependability of the software they produce, as software is becoming the dominant source of competitive differentiation in organizations’ products and services. Simultaneously achieving and improving agility and dependability will be one of the biggest challenges for 21st century software engineers.

When I was working on the NASA High Dependability Computing program with Vic Basili and others, we found that one of the biggest challenges was just defining “dependability.” There were several cases in NASA and elsewhere in which “dependability” was equated with system mean time between failures or system uptime, but in which this emphasis led to the loss of other system properties that users were counting on. In one case, the system administrators were incentivized on uptime, and ran the system in a way the produced very high uptime, but also produced delays of up to two weeks for users depending on rapid response when requesting data or services from the system.

This indicates that when you need to produce a dependable system, it will be important to identify your most success-critical stakeholders and what value propositions that their success most depends on. Figure 2 shows that these stakeholder value propositions are likely to conflict with each other, even for the four main stakeholders in a software-intensive system. The end users’ success will depend on the system having many features whose nature and priorities they can change at any time; operating smoothly, quickly, and reliably; being compatible with their other applications; and being available right away. The sales users’ success will depend on having fancy displays that entice customers. The administrative users’ success will depend on having extensive self-monitoring, audit trails, and trend analysis capabilities.

The acquirers will have limited resources, and their success will depend on having a business case for each feature; and having a completely stabilized set of requirements. The developers’ success will depend on minimizing the risk of overrunning the budget and schedule, which will imply having the ability to make infrastructure decisions and to reuse previously-developed software. The maintainers’ success will depend on how well the system and its software are designed and structured for ease of maintenance, and how compatible they are with the other artifacts being maintained.

The red (or gray) lines in Figure 2 show actual conflicts among stakeholder dependability value propositions that were

continues on page 8
Future Challenges and Rewards

Continued from page 7.

Figure 2. Key Stakeholder Value Propositions and Their Potential Conflicts

Note: Red lines in this figure appear as gray lines in a non-color printed version.

not resolved in one of the classic failed software projects, the Bank of America Master Net system, as discussed in [Boehm et al., 2000]. The black lines were additional conflicts we found in analyzing other failed projects. The S, PD, PC, and PP annotations on the lines indicate whether a line primarily reflected conflicts among the project’s success models, product models, process models, or property models. One implication of the diversity of model clashes on failed projects is that current “model-driven development” initiatives need to consider more than just product models. A recent empirical study on model clashes by one of our Ph.D. graduates found that product-product model clashes were only about 30% of the total number of model clashes found [Al-Said, 2003].

The fact that there are so many potential conflicts among even the four main success-critical stakeholders means that there are many potential win-lose situations that a project can get into. A win-lose situation usually turns into a lose-lose situation, as happened with the cancellation of the Master-Net project when the Acquirers found that their budget was being overrun, the Users were unhappy with the system’s performance, and the Maintainers were unhappy with the incompatibility of the Developer’s Prime Computer solution with their other IBM mainframe applications.

In such a situation, it is important to manage stakeholders’ expectations, invest more effort into ensuring feasibility of the proposed solution with respect to all the stakeholders and to negotiate a mutually satisfactory or win-win set of specifications, plans, and resources before proceeding into development. When negotiating requirements, it is better in the early stages to replace words with non-negotiable connotations such as “requirements” (things claimed by right and authority) by words like “objectives” and “goals.”

Finally, in order to converge on a software-intensive system that is mutually satisfactory to a diverse set of stakeholders in such areas as banking, medicine, public services, manufacturing, transportation, or commerce, it is increasingly important for more and more software engineers to be knowledgeable not only with software concepts and techniques but the concepts and techniques of the organizations using the software. In the 21st century and beyond, it is people that understand the tradeoffs between software feasibility and applications-domain feasibility who will be most influential in architecting the software-intensive systems of the future.

continues on page 9
Future Challenges and Rewards

Continued from page 8.

Diversity

The NASA example above indicates that a value-neutral, one-size-uniformly-fits-all (OSUFA) definition of dependability as system uptime is often unlikely to succeed for other success-critical stakeholders. This conclusion becomes even more pronounced in considering the trends towards cross-cultural globalization described in such books as The World Is Flat [Friedman, 2005]. An example in the area of software capability maturity models is that in Thailand, only 17 of 380 software-developing companies decided to use the U.S.-developed Software CMM, and only 3 of the 17 companies proceeded to increase their CMM level. Much of the difference in usage can be explained by the differences in the more individualistic, masculine, short-term oriented U.S. culture and the more community-oriented, feminine, long-term oriented Thai culture, as shown in Table 1 [Hofstede, 1997].

Some other cultural differences that cause difficulties for OSUFA practices are provided in [Hall, 1976]. One example is the difference between polychromatic cultures, in which interruptions during task performance such as blinking e-messages and pop-up windows are welcomed; and monochromatic cultures in which such interruptions cause frustration as people try to focus on the closure of one task at a time. Hall also found that contracts in a high-content culture such as the U.S. averaged 10 times as long as contracts in a high-context culture such as France.

OSUFA contracting practices and process standards are a particularly poor match to another 21st century trend toward large software-intensive systems of systems (SISOS). These add scalability to the simultaneous challenges of achieving high agility and high dependability discussed above. Figure 3 compares the average change request processing times for two recent large-scale software-intensive systems of systems using OSUFA, high-content contracting instruments and processes. Having to deal with many contract-affected changes that average 141 workdays to achieve closure is not a recipe for success in developing change-responsive and on-schedule large SISOS.

Interdependence: No Man, Artifact, or System Is an Island

In the large-scale software-intensive systems of systems (SISOS) discussed above, the component systems no longer have the luxury of operating autonomously. They act on information from other SISOS elements and supply information for them. Where, before, they may not have been concerned with such issues as interoperability and network security, now they are necessary considerations. Each component system is no longer an island, entire of itself, and the SISOS is not an island either. Many of them must interoperate with over 100 separately evolving systems, with which they need to stay synchronized.

But beyond this, it is important to recognize that this applies not only to the SISOS and its elements, but also to the processes, methods, and tools used to define, design, develop, deploy, and evolve it. No set of requirements, designs, or plans is an island. Nor can software engineering, hardware engineering, or human factors engineering be considered as islands. They all need to be engineered concurrently. And this concurrency needs to be incrementally synchronized and stabilized to avoid divergence, chaos, or analysis paralysis. Figure 4 provides one vision for how to do this, based on a recent National Research Council study on human-system integration [Pew and Mavor, 2007].

The recent Software Engineering Institute report on Ultra-Large-Scale Systems [SEI, 2006] makes a good case that such SISOS are best considered as ecosystems, which contain internal ecologies among their component systems, and which participate in external ecologies within which they need to stay viable. The report includes numerous good recommendations for improving SISOS acquisition, requirements engineering, design, multidimensional and dynamic quality assurance, and evolution.

Table 1. Cultural Differences Between the U.S. and Thailand [Hofstede, 1997]

<table>
<thead>
<tr>
<th>Country</th>
<th>Power Distance</th>
<th>Uncertainty Avoidance</th>
<th>Individualism</th>
<th>Masculinity</th>
<th>Long term Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index</td>
<td>Rank</td>
<td>Index</td>
<td>Rank</td>
<td>Index</td>
</tr>
<tr>
<td>US</td>
<td>40</td>
<td>38</td>
<td>46</td>
<td>43</td>
<td>91</td>
</tr>
<tr>
<td>Thailand</td>
<td>64</td>
<td>21-23</td>
<td>64</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

continues on page 10
A further important point in defining this kind of ecosystem is that considerably more effort is needed up front to avoid getting its concurrently-engineered requirements, designs, and plans pointed toward an ecological crisis. Increasing interdependence and the rapid pace of change mean thinking about not just the first derivative but also the second and maybe third derivatives. The best book I’ve read recently is “The Logic of Failure” [Dorner, 1996]. It shows through a wealth of examples and ecology simulation games that people with their built-in fight-or-flight survival skills have a horrible track record in managing ecologies, but that they can get better at managing ecologies with practice and with better tools for understanding side effects.

The need for more up-front investment is corroborated by the analysis summarized in Figure 5, [Boehm-Turner, 2004]. It shows that the “sweet spot” for “how much up-front architecting and risk resolution effort is enough” increases as the size of the SISOS increases. This effort is not just in writing specifications and plans, but more importantly in exercising models, simulations, prototypes, and partial SISOS implementations before committing to go forward into SISOS development.

### Conclusions

In the 21st century and beyond, software will provide the sensing, communications, and decision support capabilities that enable people to become aware, to understand, and to collaborate in addressing the problems and opportunities they will have from local and personal levels to global levels. At each level, software capabilities will strongly determine how well people will be able to understand each other and come together to find ways to make their local and global situations more mutually satisfactory and sustainable into the future.

If this software and the collaboration it supports are done well, the world could have a Golden Age going well beyond the dreams of the Athenian and Renaissance philosophers. If it is done poorly, it will exacerbate tensions and mistrust, obfuscate mutual understanding, and cut off many of the options for joint gain that enable people to negotiate mutually satisfactory and sustainable agreements.

This is not to say that software is a silver bullet for resolving social problems. Some of the limits to software perfectability are imposed by limits to social perfectability. An example is provided by Conway’s Law, which states that the structure of a software product reflects the structure of its sponsoring and development organizations. The converse of Conway’s Law then states that we will be able to create perfect software as soon as we learn how to create perfect organizations – which seems unlikely to happen soon [Boehm, 2007].

But it is to say that making incremental improvements in developing better software will have significant leverage in incrementally making the world a better place. And it is to say that being a software engineer in the 21st century will be one of the most challenging and rewarding careers available to people. The challenges we have discussed above of simultaneously dealing with uncertainty and emergence, rapid change, dependability, diversity, and interdependence will often be formidable, but will also be opportunities to make significant contributions that will make a difference for the better.

---

*Figure 3. Average Change Request Processing Times for Two Large SISOS: Workdays*

![Figure 3](image-url)
Future Challenges and Rewards

Continued from page 10.

<table>
<thead>
<tr>
<th>General/DoD Milestones</th>
<th>ICM Lifecycle Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>System scoping</td>
<td>Levels of activity</td>
</tr>
<tr>
<td>Understanding needs</td>
<td></td>
</tr>
<tr>
<td>Envisioning opportunities</td>
<td></td>
</tr>
<tr>
<td>Goals/objectives</td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
</tr>
<tr>
<td>Architecting and designing solutions</td>
<td>a. integration</td>
</tr>
<tr>
<td></td>
<td>b. human</td>
</tr>
<tr>
<td></td>
<td>c. hardware</td>
</tr>
<tr>
<td></td>
<td>d. software</td>
</tr>
<tr>
<td>Life-cycle planning</td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
</tr>
<tr>
<td>Negotiating commitments</td>
<td></td>
</tr>
<tr>
<td>Development and evolution</td>
<td></td>
</tr>
<tr>
<td>Monitoring and control</td>
<td></td>
</tr>
<tr>
<td>Operations and retirement</td>
<td></td>
</tr>
<tr>
<td>Organizational capability improvement</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Representative Levels of Effort by Phase

References


continues on page 12
Future Challenges and Rewards

Continued from page 11.

12. Software Engineering Institute, Ultra-Large-Scale Systems, CMU/SEI, Pittsburgh, PA, 2006

About the Author

Dr. Barry Boehm received his PhD in Math from UCLA in 1964. He is currently a Professor at the University of Southern California. His contributions to the field include the Constructive Cost Model (COCOMO) family of systems and software engineering estimation models, the Spiral Model and Incremental Commitment Model of the systems and software engineering process, and the Theory W (win-win) approach to systems and software management and requirements determination. He has received the ACM Distinguished Research Award in Software Engineering and the IEEE Harlan Mills Award, and an honorary ScD in Computer Science from the University of Massachusetts. He is a Fellow of the primary professional societies in computing (ACM), aerospace (AIAA), electronics (IEEE), and systems engineering (INCOSE), and a member of the U.S. National Academy of Engineering. He has authored a multitude of articles, books and papers.

Dr. Boehm served within the U.S. Department of Defense (DoD) from 1989 to 1992 as director of the DARPA Information Science and Technology Office and as director of the DDR&E Software and Computer Technology Office. He worked at TRW from 1973 to 1989, culminating as chief scientist of the Defense Systems Group, and at the Rand Corporation from 1959 to 1973, culminating as head of the Information Sciences Department. He entered the software field at General Dynamics in 1955.

Author Contact Information
Email: boehm@usc.edu

Figure 5. Large SISOS Need More Time for Architecture and Risk Resolution
Are you getting **All** you can from your software investment?

**The DACS ROI Dashboard**

Impact on Quality (% defect reduction)

- **75th Percentile:** 90%
- Oklahoma City Air Logistics Center, Test Program Set and Industrial Automation line observed the defect density decrease 91%, from 3.3 to 0.3 defects per KSOLOC, while achieving CMM Level 4.

**Technologies Covered:**
- SEI CMM/CMMI
- SEI Team Software Process (TSP)
- SEI Personal Software Process (PSP)
- Inspections
- Reuse
- Cleanroom
- And Many More!

**Graphs Showing Impact Of Software Technologies On:**
- ROI
- Productivity
- Quality

**Summarizes Facts From Open Literature**

Data & Analysis Center for Software
P.O.Box 1400
Rome, NY 13442-1400

http://iac.dtic.mil/dacs

Access the DACS ROI Dashboard!
http://www.thedacs.com/databases/roi/
Addressing the Software Engineering Challenges over the Years and into the Future

Dr. Kenneth E. Nidiffer
Software Engineering Institute
Carnegie Mellon University

The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man’s handiworks – F.P. Brooks [1]

Introduction

The field of software engineering is exciting to me because there are so many new and interesting challenges. These challenges have been created by the need for ever-increasing growth and complexity of software-intensive systems over the last 50 years. Norman R. Augustine, past chairman of Lockheed Martin Corporation and five-time recipient of the Department of Defense’s Distinguished Service Medal, provided an insight to this complexity when he noted in one of his laws that “software grows by an order of magnitude every 10 years”. [2] Figure 1 presents this growth in terms of aircraft software. [3]

Of interest also is the percent of the pilot’s functions supported by software. A chart from the Defense Systems Management College showing the growth in percentage of aircraft requirements involving software control, illustrates an increase from 8% for the F-4 aircraft (developed in the early 1960s) to 80% for the F-22 aircraft (developed in this decade). In very few fields has the complexity and intricacy increased so fast. [4] For example, take the construction industry where, compared
Addressing the Software Engineering Challenges

Continued from page 14.

to software, the interfaces are well defined and visible. In the 1960’s a one-story building could be easily built and in 1970 ten-story buildings were also being easily constructed but then we have to stop. There are only a few 100-story buildings built and no attempt is being made to build a 1000-story building let alone a 10,000 story building or higher.

In a report lead by Linda Northrop, Director of Product Line Systems at the Software Engineering Institute, entitled “Ultra-Large-Scale (ULS) Systems – The Software Challenge of the Future”, the author team points out that fundamental gaps in our current understanding of software and software development at the scale of ULS systems present profound impediments of the technically and economically effective achievement of the DoD’s goal of deterrence and dominance based on information superiority [5, 6]. This article presents some thoughts based on my almost 50 years in the field on the history and future of software engineering – looking at software engineering through the lens of a set of real-world challenges and the lessons learned over the years that are shaping and limiting our future progress in meeting our national goals.

An observation of mine over these years is that there are points in time where software engineering reaches bottle necks or plateaus where it becomes difficult to develop the next level of complex system and at these bottle necks we continue to see a similar set of problems attributed to software engineering and its associated disciplines. As we struggle to develop software-intensive systems at or near these plateaus thought leaders emerge to provide the software-engineering community with new ways of thinking about how to approach the development of systems at the next level of complexity. We incorporate these different ways of thinking into the field of software engineering and develop new processes, methods and tools to move us forward. This new set of actions serve us well until we reach the next plateau, which again requires thinking differently about how we develop the next level of complex system. This article discusses these different plateaus starting with the 1960’s.

How We Got Where We Are Today

The problems presented by software have long been recognized and by the late 1960’s it was clear that the software problem was real, significant, growing and would be with us for a long time; thus, the need existed to do something to mitigate the effects of this enabling technology. The term “software engineering” was first used as the provocative theme of workshops held in Garmisch, West Germany in 1968 that focused on the growing problems presented by software. For the purpose of this paper the working definition of “software engineering” is the technological and managerial discipline concerned with the systematic production and maintenance of software-intensive systems that are developed and modified on time and within cost and quality estimates. With respect to this definition, software engineering includes computer science, economics, management sciences, communication, social engineering and the engineering approach to problem solving. [7]

Software engineering is a pragmatic discipline that relies on computer science to provide scientific foundation in the same way traditional engineering disciplines such as electrical engineering and chemical engineering rely on physics and chemistry. Software is intangible in that it has no mass, no volume, no color, no order --- no physical properties. Because software has no physical properties, in theory, software is not subject to the physical laws such as Newton’s laws or Maxwell’s equations. Furthermore, even though in practice the software is susceptible to the laws of physics (e.g. transmission delays, execution time, gaps between event and responses to that event, etc,) we do not adequately account for these in software and tend to discount them as not significant. This intangibility, lack of physical properties and discount to execution times for software limits the number of fundamental guidelines and basic constraints available to shape the architecture, design and implementation of the product. The excessive degrees of freedom are both a blessing and a curse to software engineering. [8] As a result, a software engineer is encouraged to think differently. Grace Murray Hopper, the first lady of software, used to say “if you tell me that you are going to do something because that is the way we have always done it, I am going to materialize before you and haunt you for a whole year”. She was not saying that you do not want to build on the past successes, but she was saying that which got you though so far will not necessarily get you through the next level of complexity.

The software engineering community has matured as larger and more complex software-intensive systems have been built. However, in some sense we are losing the history of software engineering and it not clear if the community really cares even though it should, due to the importance of lessons learned in building new systems. For example, I have worked with a significant number of wonderful people who have led us through the software revolution to date and are gone or may/ could retire in the near future, such as, V. Basili, B. Boehm, G. Booch, F. Brooks, T. DeMarco, R. Fairley, G. Hopper, W. Humphrey, B. Ireland, J. Ishbiah, T. McCabe, H. Mills, L. Aviation and the Space program.
Addressing the Software Engineering Challenges

Continued from page 15.

Putman, R. Thayer, D. Reifer, W. Riddle, W. Royce, M. Shaw, A. Wasserman, J. Weinberg and R. Wolverton. Furthermore, the software engineering community has not been good about documenting its legacy as attested by the limited number of references in the technical literature before 1980. It has also been my experience in teaching graduate software engineering for over 20 years that the students are much more interested in understanding the latest technology than learning from the past. George Santayana, a noted philosopher, spoke this half-truth warning: “Those who cannot remember the past are condemned to repeat it - that is, they do not remember failures and are likely to repeat them and they do not remember successes and are less likely to repeat them”. We are at an interesting quandary today about which of the traditional software engineering wisdom is still applicable to the current situation and which of the new process, methods and tools are a good fit. Table 1 provides some software-intensive system trends that identify some of the changes in the environment facing the developer of software-intensive systems.[9, 10] The following paragraphs provide a very short overview of where we have been from the 1950’s to the 1990’s and some lessons learned.

In the 1950’s, we treated software like hardware. We developed software flow-charts on drafting tables that were often isolated from the so-call “real work” that was being accomplished. We learned that this work environment did not produce good results as the software became more complex; it caused the need for increased communications with other members of the development team. In the 1960’s, software engineering matured into craft industries often led by “cowboy” programmers and engineers as heroes, which, in turn led to a “code and test” life cycle development process. For example, it was not uncommon to have an operational system, especially satellite systems, be supported in real-time by large teams of software engineers developing and uploading software patches. We learned from this experience that software is changeable up to a point when it becomes difficult to change anymore; that software and hardware reliability are a different phenomenon in that local fixes in software can cause global problems; to avoid cowboy programming due to dependence on the cowboy; and to respect software and hardware development differences. Fred Brooks’ book, The Mythical Man-Month: Essays on Software Engineering, highlighted many of these differences. [11] We also learned from this experience that both the acquirer and developer needed to obtain more visibility into the entire software engineering life cycle as software continued to represent more of the systems functionality.

At the beginning of the 1970’s, due to problems not adequately addressed in the 1960’s, we observed several un-maintainable and un-diagnosable systems. At this point in time, Winston Royce introduced his “Waterfall Model” to the community based, in part, on his observation of Benington’s definitional work on the successful Semi-Automatic Ground Environment (SAGE) air-defense command and control software development process in 1956.[12, 13] Royce’s model was an instant success in terms of the acquirer understanding of

Table 1. Some Software-Intensive System Trends

<table>
<thead>
<tr>
<th>Previous</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone systems</td>
<td>Everything connected (maybe)</td>
</tr>
<tr>
<td>Relatively stable requirements</td>
<td>Rapid requirements change</td>
</tr>
<tr>
<td>Requirements determine capabilities</td>
<td>Commercial off-the-shelf (COTS) capabilities</td>
</tr>
<tr>
<td></td>
<td>determine some requirements</td>
</tr>
<tr>
<td>Platform centric</td>
<td>Enterprise centric</td>
</tr>
<tr>
<td>Enough time to keep development stable</td>
<td>Ever-decreasing cycle times</td>
</tr>
<tr>
<td>Dominate Prime</td>
<td>Strategic Teaming</td>
</tr>
<tr>
<td>Failures locally critical</td>
<td>Failures broadly critical</td>
</tr>
<tr>
<td>Completely defined systems with specific</td>
<td>Complex, adaptive emergent systems of systems</td>
</tr>
<tr>
<td>functionality</td>
<td></td>
</tr>
<tr>
<td>Repeatability-oriented process, maturity</td>
<td>Adaptive process models</td>
</tr>
<tr>
<td>models</td>
<td></td>
</tr>
</tbody>
</table>

continues on page 17
Addressing the Software Engineering Challenges

Continued from page 16.

how to assess the developer’s software engineering activities of the software development life cycle. The model extended the work of Benington by introducing the notion of explicit feedback among the stages of software development process; the types of documentation needed at each stage and the need to do development at least twice - the first pass could be viewed as a prototype. The idea of “do it twice” is often overlooked in terms of Royce’s contributions. At this same period of time Harlan Mills, Michael Fagen and others began to introduce more formalized methods for software development.[14, 15] Most of the contributions were well received; however, they became, at times, hard to apply as the software grew in complexity. For example, most were heavy with respect to documentation which was difficult to review and maintain. In sum, the 1970’s taught the software engineering community to better define software engineering life cycles; to appreciate the benefits of prototyping and phased development; to eliminate errors early in the life cycle to avoid downstream costs; to determine the systems purpose before finalizing the design of the software; to focus on the software engineering development environment; and to understand the high cost of software support versus development - support for software was running about 60% of the overall life cycle costs.[16] The 1970’s also produced one of the first essays on the human element entitled: Psychology of the Computer Programmer, by Jerry Weinberg, which was the first book to address programming as an individual and team effort.[17]

Various processes and methodologies were developed and refined in the 1980s and 1990s to “tame” the software crisis, with varying degrees of success. However, it was and currently is widely agreed that there is no “silver bullet” - that is, no single improvement or change in approach which will prevent project overruns and failures in all cases. In general, software projects which are often large, complicated, and involve unfamiliar aspects are still particularly vulnerable to large, unanticipated problems. In the 1980s, the term “software crisis” was again put forth to describe the impact of rapid increases in computer power and the complexity of the problems which could be tackled. The causes of the software crisis were linked to the overall complexity of the software process and the relative immaturity of software engineering as a profession. The crisis manifested itself in several ways:

- Projects running over-budget.
- Projects running over-time.
- Software was of low quality.
- Software often did not meet user needs.
- Projects were unmanageable and code difficult to change.

In the light of real problems nature abhors a vacuum and thus responds. The software crisis created several positive initiatives, such as, the formation by the Department of Defense of the Software Engineering Institute with the mission of improving the state of software engineering practice and industry’s response; the formation of the Software Productivity Consortium with the same mission; the Defense Acquisition University’s development of the Software Acquisition Management Course; the formation of the Microelectronics and Computer Technology Company which looked at both computer and software technologies; the formation of the Fraunhofer Institute for Experimental Software Engineering; the creation of the Center for Software Engineering at the University of Southern California; and the emergence of several graduate-level university programs in software engineering. It also produced several important Department of Defense studies to address these software issues and do something about them. [18, 19]

Numerous new standards, processes, methods and tools were introduced in the 1980’s and 1990’s that improved software engineering. Some examples are Smalltalk and C++ to support reuse and object oriented development; modeling languages such as UML to support model-based development; Ada programming language to support real-time processing; COCOMO and other cost estimating relationships to support software estimation; refinement of earned-value techniques to support planning and measurement; process maturity models such as CMM and CMMI to support internal process improvement development activities; Personal Software Process and Team Software Process to improve the quality and productivity of developing software; Plan-Do-Check-Act cycle and the IDEAL model to improve the way we introduce and measure change; Spiral Model and the Rational Unified Process for reducing development risk; product line development to limit rework and increase productivity; standards to codify good practices, such as ISO/IEC 12207 for software life cycle processes, ISO/IEC 15288 for system engineering life cycle processes and ISO/IEC 15939 for measurement and support of decision making; and initiatives that focus on software architecture technologies. In addition to the technologies listed many more were developed and most of these highlighted technologies provide a much broader set of benefits than listed but hopefully I have conveyed the idea that there was an increased momentum, if not an explosion, in providing supporting technologies to enable the information revolution to continue to expand in nearly all aspects of our society, including defense systems, transportation, financial

continues on page 18
services, medicine, manufacturing, and entertainment. It should be noted that this momentum is currently waning due to the significant decrease in resources going into research and development in support of software engineering technologies and their application; however, the demand is still exponentially increasing as attested by the current emergence of new technologies based on open source, agile/iterative development, distributed and internet-enabled applications, service-oriented architectures, etc.

Although it cannot be claimed that software engineering initiatives are adequately addressing all the goals of the Department of Defense or of society as a whole, we have learned a lot more with respect to the development of complex software intensive systems. In the 1980’s and the 1990’s we have learned or better yet are still learning today that there are many roads to increasing productivity, improving our processes, increasing individual productivity, reducing rework, decreasing our development cycle times, becoming more adaptable and timely in the delivery of capabilities, understanding the necessity and benefits of developing architectures that exhibit good quality attributes, unleashing the power of service oriented architectures and web oriented technologies, becoming more considerate of all our stakeholder needs, and understanding that software engineering is of the people, by the people and for the people! [20]

Are we where we need to be?

Although we are improving our software engineering capabilities we are not necessarily winning the race. Reports such as the Standish Group’s document an unimpressive record in large software projects in the private sector. In the 2003 Standish Group’s report only 34% of all projects were deemed to be successful. [21] Although this success rate is considered a significant improvement from the 1994 rate of 16%, most would agree that this rate is inadequate! The challenges associated with developing ever increasing software intensive systems are still with us and the road ahead is not for those who are faint of heart as attested by the results of the latest significant effort to understand where we are and where we should be going in the area of software engineering which is discussed next.

The Deputy Under Secretary of Defense for Acquisition and Technology hosted the Defense Software Summit in Washington, DC, on October 18–19, 2006. The purpose of the summit was to identify the current situation, issues, barriers, and recommendations concerning Department of Defense software development. The summit provided a forum for guest speakers to present information and for attendees to conduct workshops. [22]

The summit raised awareness of significant software engineering and management issues within the Department. Summit participants helped identify specific actions the Department needs to take to reduce or eliminate adverse situations and to solve problems. A key perspective of the summit was that improved systems and software engineering methods may reduce problem root causes and provide $24B in cost avoidance over the DoD Five-Year Defense Plan. The problem root causes include lack of requirements discipline, limited staff experience, external adverse influences, poor planning, resource and budget constraints, weak contract management, funding turbulence, inadequate program documentation, and incomplete risk management. The workshop results focused on identifying current software engineering practice issues and barriers (see Table 2) and providing software engineering practice recommendations – all of which are summarized below.

Software engineering practice recommendations identified were:

- Translate best practices into policy where needed
  - Revise old standards and update as needed
  - Review contracting guidelines to ensure currency and accuracy
  - Improve system engineering planning, introducing guidance resembling that of a computer resource life cycle management plan
  - Ensure continuity of engineering personnel for large-scale projects to reduce staff turbulence
  - Conduct independent program assessments to ensure that the program is following best practices
  - Encourage industry to request independent assessments as well
  - Team software and system engineers within a program

- Provide a cadre of experts to:
  - Assist project start-ups to help balance performance schedule and cost among hardware and software performance, schedule, cost, supportability, security, and risk
  - Provide architecture expertise to support programs
  - Provide technical advice to help balance perspectives such as system, system of systems, family of systems,

continues on page 19
Addressing the Software Engineering Challenges

Continued from page 18.

Table 2. Software Engineering Practice Issues and Barriers

<table>
<thead>
<tr>
<th>Issues</th>
<th>Barriers</th>
</tr>
</thead>
<tbody>
<tr>
<td>The impact of system requirements upon software is not consistently quantified and managed in development or sustainment.</td>
<td>DoD, contractor, and project management office cultures or mind-sets are hard to change, causing clashes between groups and functional discipline areas.</td>
</tr>
<tr>
<td>Conditions leading to software requirements changes are not clearly understood.</td>
<td>Resource limitation: Many good practices are sidestepped because of limited dollar resources.</td>
</tr>
<tr>
<td>Fundamental system engineering decisions are made without full consideration of software engineering.</td>
<td>Developers lack the availability of validation of tools, such as Underwriters Laboratory.</td>
</tr>
<tr>
<td>Software life cycle planning and management are ineffective; acquirers and suppliers need better software life cycle planning and management methods.</td>
<td>Developers also lack incentives to take intelligent steps to solve government and contractors’ software problems.</td>
</tr>
<tr>
<td>The quantity and quality of software engineering expertise are insufficient to meet the demands of the government and the defense industry.</td>
<td>The acquisition community does not seem to fully understand the incentive structure of the acquisition system.</td>
</tr>
<tr>
<td>Traditional software verification techniques are costly and ineffective for dealing with the scale and complexity of modern systems.</td>
<td>There are constraints on funding, such as congressional funding restrictions.</td>
</tr>
<tr>
<td>There is a failure to ensure correct, predictable, safe, and secure execution of complex software in distributed environments.</td>
<td>Misalignment between program and contractor goals will cause severe problems.</td>
</tr>
<tr>
<td>Inadequate attention is given to total life cycle issues for COTS/NDI impacts on life cycle cost and risk.</td>
<td>Leadership is still focused on hardware; many Defense Acquisition University courses are devoid of software; and there is a perspective that software is magic.</td>
</tr>
<tr>
<td>More software expertise needs to be applied to planning and management.</td>
<td>The acquisition community has a diminishing experienced software workforce.</td>
</tr>
<tr>
<td>Software personnel staffing is inadequate across the system life cycle.</td>
<td>Personnel procedures prevent a rapid transition of skilled staff to new programs.</td>
</tr>
<tr>
<td>Use of standard terminology needs improvement. There is too much jargon. Need a clear definition of “system of systems.”</td>
<td>Acquisition methods have insufficient infrastructure (e.g., guidelines, procedures, and incentives) to create and facilitate reuse across organizations.</td>
</tr>
<tr>
<td></td>
<td>Many leaders do not understand software or appreciate its challenge.</td>
</tr>
</tbody>
</table>

continues on page 20
Addressing the Software Engineering Challenges

Continued from page 19.

net-centricity, network, data sharing, interoperability, security, and transparency, among others.

- Resource training and education to:
  o Realistically create experts to satisfy personnel needs
  o Provide leadership training to enable informed decision making regarding software
  o Reinforce software engineering and management in the systems, planning, development, research, and engineering acquisition specialty
- Revise contracting methods and procedures to:
  o Allow additional detail for software requirements if needed
  o Raise software as a major factor and improve visibility within the work breakdown structure for mission-critical or software-intensive systems
  o Clarify contractor role in conducting system engineering technical analysis for software integration
- Create or identify software tools to help manage the software development process:
  o Create a DoD architecture framework that better represents software needs
  o Promote the use of product-line practices by acquisition programs and industry for the creation and use of cross-program capabilities and assets. Examine associated lessons learned regarding reuse
  o Identify tools to support decomposing large programs or architectures into smaller ones
  o Provide a DoD test bed or laboratory that provides persistent network test environments
  o At design reviews, encourage demonstrations; have developers show a capability or requirements satisfaction

It has been argued that these current problems are similar to those we have faced before at previous plateaus; thus, the community is only addressing symptoms of the real problem versus the cause which requires a new way of thinking about how we will develop complex software-intensive systems at the next level of complexity.

The road ahead – are we going in the right direction?

Since 1968 when the community agreed to characterize software development as an engineering problem, tremendous progress has been made in the field of software engineering. Although we are still faced with these classic problems that we have had for a long time as attested by the most recent problems identified, our new software-engineering technologies have allowed us to advance the state of practice. The net effect has made it possible to construct ever increasing complex, software intensive systems. As our capabilities in software engineering have increased so has our vision and desire to build more complex systems – the challenge being that our vision and aspirations continue to exceed our capabilities. Action plans are currently being put in place to address the current set of software engineering gaps as identified at the Summit and other initiatives. It is acknowledged that there are limited resources and thus, the need to focus on what can be done; however, it is important not to lose site of DoD’s goal of deterrence and dominance based on information superiority.

The jury is still out with respect to the correctness of our direction in the long term. There is, without question, a critical need for a significant increase in engineering research and applied development of the appropriate technologies for us to advance the state of systems engineering to meet the needs of tomorrow. I am reminded about the significant difference in software engineering to other engineering technologies – the excessive degrees of freedom and its inclusion of so many other disciplines. The degrees of freedom have now been further increased as the networks are the computer systems of the future. In turn, software engineering will play a significant role as the scale of our new ultra-large systems and systems of systems limit our ability to maintain hierarchical control and move us to a decentralized control environment where wide-band communication, visualization, adaptive system and human-factor/human-interaction technologies become more dominate in the field of software engineering. Assuming DoD’s goal and Augustine law continue to hold, a significant degree of difference in scale will exist for systems of the future. The following observation was presented in Ultra-Large-Scale (ULS) Systems – The Software Challenge of the Future report: “The difference in scale between today’s largest systems and systems that we will build in the future is like the difference between constructing buildings and guiding the growth and evolution of cities.” [23] My sense is that there is a need to change the perspective on how software engineering is characterized if the problems we will face are to be effectively addressed. I welcome your thoughts to extend the dialogue about innovative ways to consider and apply software engineering.

continues on page 21
Addressing the Software Engineering Challenges

Continued from page 20.

References

3. Sloane, Stan EVP, Lockheed Martin, Presentation at the Software Productivity Consortium’s Executive Round Table, 2004

About the Author

Dr. Kenneth E. Nidiffer has over forty-five years of experience in the marketing, research, development, maintenance, and acquisition of software-intensive systems. He has held several executive-level positions in the Department of Defense and industry (e.g. Software Productivity Consortium, Northrop Grumman Corporation and Fidelity Investments) where he has managed software-intensive programs and led applied research efforts to improve the state of software engineering practice. Dr. Nidiffer is currently Director of Strategic Plans for Government Programs at the Software Engineering Institute, Carnegie Mellon University.

Author Contact Information

Phone: 703.908.1117 (Direct)
E-mail: nidiffer@sei.cmu.edu.
BACKGROUND

For decades, Navy C4I program offices have delivered complex technology to meet Navy and Joint warfighter requirements. The capabilities they provided were delivered more rapidly than large ship or aircraft programs but were rarely able to keep pace with technology that users desired. When today’s sailors upgraded their home computers to the latest version of Microsoft, they often did not see the same upgrade onboard ship for several years. When American teenagers began using commercially-provided Instant Messaging as part of their daily lives, sailors were still waiting to see an established real-time collaboration tool. And even today despite technology advances, deployed bandwidth per sailor is less than that of a typical cell phone. The average afloat network is almost seven years old and it takes years vice months to get new network technology fielded in the Fleet. Radios support only one waveform each, which means there are more than 800 variants installed in the Fleet, some of which are more than 30 years old.

In the meantime, the national threat landscape is driving the military to become lighter, more agile and better “connected.” According to the National Defense Strategy, “Uncertainty is the defining characteristic of today’s strategic environment.” The old method of providing multiple systems with dedicated hardware and applications is too expensive, cumbersome and time consuming to meet warfighting requirements.

In order to effectively support the warfighter in this dynamic and unpredictable environment, the Navy’s Program Executive Officer, Command, Control, Communications, Computers and Intelligence (PEO C4I) has embarked on an aggressive plan to change not only its technical model, but also its business approach. The principles of naval open architecture form the basis for this multifaceted strategy. The 12 December 2006 Navy Rhumblines publication defined open architecture as, “… the confluence of business and technical practices yielding modular, interoperable systems that adhere to open standards with published interfaces. This approach significantly increases opportunities for innovation and competition, enables reuse of components, facilitates rapid technology insertion, and reduces maintenance constraints. Open architecture delivers increased warfighting capabilities in a shorter time at reduced cost.” PEO C4I program managers and staff are evaluating and leveraging commercial trends toward standardization, consolidation and service-oriented architecture (SOA) while simultaneously exploring opportunities for increased vendor competition and capturing government data rights. Through this approach, PEO C4I will gain improved access to innovative technologies and be able to rapidly deploy enhancements.

PEO C4I has developed a comprehensive plan to become more responsive to Fleet readiness requirements, and to increase supportability, standardization, system interoperability, network security and Joint alignment. In order to achieve this overarching goal of providing a more agile, integrated C4I capability for the Navy, PEO C4I will:

• Reduce the number of servers and applications while increasing server utilization
• Transform application programs into community of interest providers
• Increase bandwidth capacity and utilization
• Implement a C4I rapid capability development and fielding process similar to the Navy Submarine community’s Acoustic Rapid COTS Insertion model
• Capitalize on acquisition innovation, leveraging processes and concepts such as Rapid Development Capability (RDC) programs and the Federated Development and Certification Environment, both described in more detail later

TECHNICAL APPROACH

PEO C4I’s portfolio consists of nearly 150 programs and projects across 11 program offices. They vary in size, funding and customer base but generally fall under one of three categories: Communications, Network Infrastructure and Command & Control and Intelligence Applications. In each of these areas, PEO C4I has devised a technical strategy to migrate these programs to a more open, capability-focused portfolio vice individual systems with complex interface specifications and interoperability challenges. The following sections describe the technical approach for each of these areas that PEO C4I is pursuing in order to achieve the strategic goal of providing common, interoperable capabilities more rapidly at reduced costs, aligning with the Naval OA Strategy.

Communications

PEO C4I’s communications programs are divided into two primary areas: Satellite Communications (SATCOM),
both military and commercial, and Tactical Communications, consisting of a host of radio and ancillary products across the frequency spectrum. In both cases, one of the most significant complexities and cost drivers is the number of variants fielded, which require support, spare parts and expertise to maintain the aging products. The Navy’s Communications Program Office has embarked on a long-range plan to migrate its portfolio to more integrated solutions, focusing on fewer multiband, software defined, Internet Protocol based products.

The future developmental program in the military SATCOM realm is the Navy Multi-band Terminal (NMT), which is expected to begin replacing legacy super high frequency, extremely high frequency and global broadcast system terminals in the FY2012 and beyond timeframe. NMT provides a single terminal for wideband (Ka, X-Band) and protected (Q/Ka) satellite communications.

The primary Commercial SATCOM products, Commercial Wideband Satellite Program and the International Maritime Satellite (INMARSAT) system, will migrate to the Commercial Broadband Satellite Program (CBSP), which is one of PEO C4I’s most aggressive RDC programs. CBSP will provide additional bandwidth to meet the required level of SATCOM per platform and full redundancy in the event military SATCOM is unavailable. A new-start program in FY2008, CBSP has received additional funding to expedite delivery and will begin fielding next year.

For Tactical Communications, PEO C4I is implementing a medium- to long-term strategy that leverages an existing production program called Digital Modular Radio (DMR), and eventually migrates to DoD’s Joint Tactical Radio System (JTRS). With many tactical radios aging beyond their useful life before JTRS Initial Operating Capability, PEO C4I intends to refresh over the next four years with DMR units and then begin to field JTRS radios in the FY2012 and beyond timeframe. Both radio systems will provide software-reprogrammable, multiband/multimode radio capability.

Overall, the primary goal of the communications technical roadmap is to neck down to a limited number of preferred products in order to achieve a more manageable portfolio. The benefits of this strategy range from fewer configurations to support, reduced shipboard footprint and increased capability, to include multiple waveforms per radio and higher overall data rates per platform.

Network Infrastructure

Over the last two decades, the explosion of networking capability has created unintended consequences aboard afloat platforms. For each type of network requirement the Navy identified (e.g., tactical, administrative, classified, coalition, etc), a separate, distinct network was developed and installed. As a result, the Navy’s program office for Networks, Information Assurance and Enterprise Services manages a portfolio of multiple, unique networks with various classification levels, operating systems and protocols. These individual networks are difficult to certify and defend from attacks; they

Figure 1. Pearl Harbor, Hawaii (Jan. 24, 2007) - Electronics Technician 2nd Class Joseph Melchor, assigned to the Arleigh Burke-class guided-missile destroyer USS Paul Hamilton (DDG 60), climbs the mast of the Paul Hamilton to perform pre-deployment maintenance on the International Maritime Satellite (INMARSAT) system. Paul Hamilton and her crew are currently making preparations to deploy as a member of the USS John C. Stennis (CVN 74) Carrier Strike Group. Stennis and its strike group are scheduled to enter 5th Fleet’s area of operation and provide support to U.S. and coalition forces operating there. U.S. Navy photo by Mass Communication Specialist 1st Class James E. Foehl (RELEASED)

continues on page 24
bring their own racks and servers; and they are unable to share server and storage resources. As a result, the space, weight and power required has reached capacity on many platforms.

In order to address these challenges, the program office developed a phased plan to migrate its primary network programs into a single overarching program called Consolidated Afloat Networks and Enterprise Services, or CANES. CANES will have at its roots primarily the Integrated Shipboard Network System, but will also incorporate the capabilities of other networks such as Combined Enterprise Regional Information Exchange System, which is used for coalition communications; the Sensitive Compartmented Intelligence Local Area Network (LAN), which is the shipboard network for special intelligence and cryptologic tactical communications; and the Submarine LAN, which is the submarine fleet’s primary network.

The basic concept of CANES is to take hardware requirements and create a single Consolidated Computing Environment using standard network infrastructure and a common rack architecture. Enterprise services will support hosting of both warfighting and administrative application programs. This evolution requires detailed technical exchanges between the programs’ engineers and a significant amount of resource reprogramming.

The expected benefits of conducting this network infrastructure transformation are:

- Reduced number of installs
- Reduced physical footprint
- Dynamic sharing of storage and processing
- More efficient use of computing power
- Improved configuration management
- Enhanced security
- Reduced non-recurring engineering costs
- Reduced manpower and training requirements

PEO C4I is working with the Navy’s PEO Enterprise Information System to achieve more consistent standards and commonality with the future shore-based network program, Next Generation Net-}

work. In addition, the program plans to leverage as much Joint capability as possible, to include the Net-Centric Enterprise Services program managed by the Defense Information Systems Agency (DISA).

Finally, the CANES solution also solves a significant configuration management and supportability problem. Currently, each ship has a different configuration of hardware and applications. Training sailors aboard ship to operate and maintain all of the systems increases the complexity and total ownership costs to the Navy. By moving to a common configuration for hardware and enterprise services, trained sailors can operate and maintain CANES regardless of ship class.

**Command & Control and Intelligence Applications**

Many of PEO C4I’s application programs currently provide not only data and services, but also enterprise services, the presentation layer and hardware as well. Each system is developed to meet its own unique requirements without considering where there may be overlaps with other programs.

*Figure 2. PACIFIC OCEAN (July 13, 2007) - Information Systems Technician Seaman Jaron Franck monitors the ship’s automated digital networking system from the radio communication center aboard Nimitz-Class aircraft carrier USS Abraham Lincoln (CVN 72). Lincoln is underway off the coast of Southern California conducting flight deck carrier qualification. U.S. Navy photo by Mass Communication Specialist 3rd Class James R. Evans (RELEASED)*

*continues on page 25*
continues on page 24.

**Figure 3.** Nias, Indonesia (Apr. 4, 2005) - Information System Technician 1st Class Josh Davis performs a communications check with a Inmarsat satellite phone system to establish communications with the Military Sealift Command (MSC) hospital ship USNS Mercy (T-AH 19) and the advance team upon arrival in Nias, Indonesia. At the request of the government of Indonesia, the Military Sealift Command (MSC) hospital ship USNS Mercy (T-AH 19) and the MSC combat stores ship USNS Niagara Falls (T-AFS 3) are on station off the coast of Nias, providing assistance as determined appropriate and necessary with earthquake disaster relief efforts and provide medical assistance to those in need. U.S. Navy photo By Photographer's Mate 2nd Class Jeffrey Russell (RELEASED)

or opportunities for efficiencies. Over time, the user communities, which have demanded capability upgrades from these stovepipe systems, rarely evaluated whether old segments were still necessary and were frustrated when the systems were unable to exchange data easily. In addition, new application programs often faced delays due to their dependence on available hardware upgrades.

PEO C4I is taking a new, aggressive approach to changing this mindset. The organization is conducting a functional decomposition of its large application programs and breaking them into community of interest services. By doing so, the necessary pieces can be pulled together into “mission-driven capabilities.” For example, if the mission at hand is a non-combat evacuation operation, the warfighter could access services such as “geo-positioning,” “decision support” and others, which would function as an orchestrated composite application, leveraging the common computing environment and enterprise services provided by CANES. This technical approach enables warfighters to obtain only the information they need for the mission vice the system pushing more information than necessary. It also enables mission and application agility by allowing users to adapt applications to changing environments and processes.

By separating the hardware from the software, PEO C4I is able to accept emerging requirements and quickly turn them into software solutions that ride on the common CANES infrastructure. Additionally, the high cost and schedule requirements of installing a system on a ship is simplified by delivering a set of CD/DVDs or providing a software download from a secure website.

continues on page 26
This methodology also makes it technically viable to reuse software, increase open competition by using a non-proprietor CANES backbone, and support other business strategies that will make C4I systems more affordable, supportable and available to the warfighter.

BUSINESS APPROACH

In order to achieve the stated goal of providing a more agile, integrated C4I capability for the Navy, PEO C4I recognizes that a change in its technical model must be combined with a change in the way it does business. Today, the C4I program offices face common barriers to achieving true speed to capability such as:

- Integration occurs at Fleet installation
- Software code is rarely reused
- Individual programs conduct individual test events
- Platforms do not have common C4I suite baselines
- Industry competition is often limited

By leveraging best practices from other communities, enlisting the support of the Space and Naval Warfare Systems Centers through an Integrated Product Team structure, and taking advantage of innovative acquisition processes such as RDCs, PEO C4I will make a significant paradigm shift in how it operates.

By moving to a common computing environment architecture with smaller, community of interest service capabilities riding on that infrastructure, PEO C4I will reduce the spiral development cycle for application programs. Using an incremental build approach, mature technologies can be rapidly fielded with lower risk. Using a concept similar to DISA’s development and certification approach, PEO C4I can achieve more rapid capability delivery by engaging the warfighter, requirements, testing and approval stakeholders earlier in the development process. This approach reduces the requirement for stovepipe test events, facilitates interoperability and migrates toward a more integrated, end-to-end testing environment. Applications can be fielded by demonstrating they meet CANES certification criteria, minimum standards compliance, security, interoperability and operational suitability and effectiveness.

Taking best advantage of existing acquisition guidance that affords a more streamlined process and more rapid delivery of capability, PEO C4I currently manages more Rapid Deployment Capability (RDC) programs than any other PEO in the Navy. As described in SECNAVINST 5000.2C, the RDC process “provides the ability to react immediately to a newly discovered enemy threat(s) or potential enemy threat(s) or to

Figure 4. ARABIAN SEA (March 6, 2007) - Information Systems Technician 2nd Class Dan Wiessman, assigned to combat systems department, checks satellite communications for voice networks aboard Nimitz-class aircraft carrier USS John C. Stennis (CVN 74). Stennis, as part of the John C. Stennis Carrier Strike Group, is on a regularly scheduled deployment in support of Maritime Security Operations (MSO). U.S. Navy photo by Mass Communication Specialist Seaman John Wagner (RELEASED)
respond to significant and urgent safety situations through special, tailored procedures...” Currently, PEO has four RDC programs in development: Commercial Broadband Satellite Program; Expanded Maritime Interception Operations Wireless Reachback; Subnet Relay/High Frequency Integrated Protocol; and Automatic Identification System to fill identified capability shortfalls.

In addition, PEO C4I is actively enforcing its Net-centric Enterprise Solutions for Interoperability (NESt) guidance, developed in conjunction with the Air Force’s Electronic Systems Command and DISA (see sidebar).

By promoting greater code reuse across the C4I community, the Navy and Joint communities can benefit from reduced development costs. In addition, in order to change the acquisition culture to better align with this strategy, PEO C4I is partnering with the Defense Acquisition University and the Navy’s PEO for Integrated Warfare Systems to develop an Open Architecture Workshop that will provide program managers, senior engineers and contracting specialists with case study investigations of how to implement OA principles. Note: [The Tactical Switching program will be the OA case study reviewed and the first class will be offered in November 2008 by Professor Kevin Corcoran, Tel: 619-524-4656, Email: Kevin.corcoran@dau.mil]

Finally, PEO C4I has also been working closely with the Navy platform PEOs to establish common C4I suite platform baselines. Within each ship class, there are often countless variations of C4I capability that results in configuration management challenges for C4I program managers. Even worse, some ship classes have opted to procure C4I capability directly from industry or other providers, which results in higher life-cycle costs when the lack of supportability, interoperability and other elements are factored in. Moving to a standard baseline of products across ship classes, the Navy will realize significant cost savings through quantity discounts, standardization of parts, training and logistics and improved refresh cycles.

The PEO C4I Open Architecture strategy is a comprehensive approach to achieve the overarching goal of providing a more agile, integrated C4I capability for the Navy by reducing infrastructure to a more lightweight, common architecture, necking down to fewer, but more capable preferred products, making better use of the capacity of those products and leveraging “business process” best practices to get capability to the warfighter quickly. In order to meet the changing face of the enemy, these fundamental changes in C4I delivery are not just good ideas; they are critical to the defense of our nation.

SIDEBAR

Net-centric Enterprise Solutions for Interoperability (NESt)

NESt is a six-volume document set that provides actionable business and technical guidance for program managers, software developers and acquisition staff to ensure compliance with net-centric, open architecture policies. It addresses software design and development, contracting language (including data rights) and provides a means for measuring compliance. Program adherence to these best practices and guidance provides the Milestone Decision Authority with the confidence that sound technical development is being executed. The URL, open to the public, is: http://nesipublic.spawar.navy.mil

POC: Mike Dettman, Associate Technical Director for Policy & Guidance
Tel: 858-537-0263,
Email: mike.dettman@navy.mil

NESt guidance also requires software programs to post their reusable software code in a repository and collaborative development website. Access to this repository is available from the NESt public site (see above) or directly at:


It is a DoD PKI-enabled site so a DoD certification is required. This site is part of a Navy-wide federation of repositories. Reuse vetting processes are executed program to program.

About the Author

Christopher A. Miller currently serves as the Program Executive Officer for Command, Control, Communications, Computers and Intelligence (PEO C41). In this capacity, he has oversight and responsibility for acquisition and life cycle management for assigned C41 programs.

He joined the PEO C41 staff in 2004 and served in the positions of Technical Director and Director of Modernization. In these roles he provided technical leadership and oversight for C41 program execution and fielding. Major accomplishments include leading a cross-service effort with the United States Air Force to establish guidance for implementing the Net-Centric Enterprise Solutions for Interoperability (NESt), which was the PEO’s first overarching guidance effort and a key enabler for delivering network-centric C4I capabilities. He also lead the development of the first consolidated fielding

continues on page 28
Navy C4I Open Architecture Strategy

Continued from page 27.

plan and Modernization Concept of Operations (CONOPS), which defined and implemented the PEO’s modernization planning, design, and execution processes.

In May 2006 Mr. Miller entered the Senior Executive Service and was selected as Executive Director/Deputy for PEO C41 and Space, where he led the program evaluation and integration efforts for all of the Navy’s C41 acquisition programs of record. He was appointed Acting Program Executive Officer PEO C41 in August 2006. Mr. Miller is a member of the Acquisition Corps, Armed Forces Communications and Electronics Association (AFCEA), and the United States Naval Institute (USNI). He has received several awards for his service including a Navy and Marine Corps Achievement Medal and the AFCEA and USNI Copernicus Award.

Prior to joining the PEO, Miller spent time at the Space and Naval Warfare Systems Command (SPAWAR), various Marine Aviation Commands and, as a consultant, numerous command and control programs for the Navy.

A native of Nashville, TN, he received his Bachelor of Arts degree from Vanderbilt University through the Naval Reserve Officer Training Program and accepted a commission with the United States Marine Corps.

Author Contact Information
Phone: 858-537-8779
Email: chris.miller@navy.mil

FREE State-of-the-Art (SOAR) Report from DACS & IATAC
(Information Assurance Technology Analysis Center)
Published July 2007 — PDF File nearly 400 Pages

SOFTWARE ASSURANCE has as its goal the ability to provide to software acquirers and users the justifiable confidence that software will consistently exhibit its required properties. Among these properties, security is what enables the software to exhibit those properties even when the software comes under attack.

This state-of-the-art report (SOAR) identifies the current “state-of-the-art” in software security assurance. It provides an overview of the current state of the environment in which defense and national security software must operate; then provides a survey of current and emerging activities and organizations involved in promoting various aspects of software security assurance; and describes the variety of techniques and technologies in use in government, industry, and academia for specifying, acquiring, producing, assessing, and deploying software that can, with a justifiable degree of confidence, be said to be secure. Finally, the SOAR presents some observations about noteworthy trends in software security assurance as a discipline.

Here is a link that will allow you to view the SOAR on the IATAC web site.

The Case for Automated Software Testing

By Bernie Gauf and Elfriede Dustin, IDT

Abstract

This article will discuss Automated Software Testing as a proposed solution to the ever increasing testing problem: This proposed solution is backed up, by presenting a background of the current testing problem, supported by results of a recent IDT survey.

Automated Software Testing refers to the “Application and implementation of software technology to allow for automation throughout the entire software testing lifecycle (STL) with the goal to improve the STL efficiencies and effectiveness.”

We discuss the importance of Automated Software Testing as part of the System Engineering Lifecycle and we describe the return on investment (ROI) of some efforts undertaken thus far, as well as other benefits. Additionally, this article will talk about some of the automated software testing pitfalls to avoid and how to accomplish successful automated software testing.

Problem: Too much time is spent on software testing!

Too much time is spent on software testing. As software programs are increasing in complexity, testing times only seem to have increased. As stated by Hailpern and Santhanam: “... debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost.”

One recent testing improvement initiative is the establishment of a task force to improve development test and evaluation. A “Memorandum for Chairman, Defense Science Board” with the subject “Terms of Reference – Defense Science Board (DSB) Task Force on Development Test and Evaluation (DT&E)”, states that “approximately 50% of programs entering Initial Operational Test and Evaluation (IOT&E) in recent years have not been evaluated...” Because of this memorandum, dated 2007, it was requested that “DSB establish a task force to examine T&E roles and responsibilities, policy and practices, and recommend changes that may contribute to improved success in IOT&E along with quicker delivery of improved capability and sustainability to Warfighters.”

Evidence of another improvement initiative appeared in the Washington Post, June 3, 2007: “The IRS has launched an initiative to enhance and expand current testing by integrating industry best testing practices to gain efficiencies that improve our overall testing processes.”

The outcome of a recent software testing survey conducted by IDT, LLC backs up the findings related to long software test timelines and high testing time percentages relative to the rest of the software engineering lifecycle. The survey’s goal was to determine software testing related issues in order to derive needed solutions, while reaching as many software testers (with a wide demographic) as possible: It was sent to tens of thousands of test engineers, posted on Quality Assurance (QA) user sites, and advertised on various Government tech sites. So far, there are nearly 280 responses from all over the world: 74% of the responses are from the U.S. and 26% of the responses are from other countries, such as India, Pakistan, Canada, South Africa, China, and Europe. More than 50% of survey respondents work for organizations of 1000 and more employees.

The survey contained various software testing related questions, and specifically the survey response to “Time currently spent on testing in relationship to overall software development lifecycle” is listed in Table 1: Almost 50% state that 30-50% of time is spent on software testing in relation to the overall software development lifecycle, and nearly 25% state that more than 50% of time is spent on it.

Automated Software Testing as Part of the System Engineering Lifecycle

Automated Software Testing success is increased, if implemented as part of the system engineering lifecycle. This includes developer involvement; starting with automated unit testing; integration-testing and then building on those initial tests, automating the system testing. Additionally, automated testing as part of the system engineering lifecycle includes stakeholder understanding of what Automated Software Test-

<table>
<thead>
<tr>
<th>Question Response Options</th>
<th>% of Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 30 %</td>
<td>28%</td>
</tr>
<tr>
<td>30 % - 50%</td>
<td>48.6%</td>
</tr>
<tr>
<td>50% - 75 %</td>
<td>14%</td>
</tr>
<tr>
<td>75% - 100%</td>
<td>7.00%</td>
</tr>
<tr>
<td>Other (please specify)</td>
<td>3.75%</td>
</tr>
</tbody>
</table>
The Case for Automated Software Testing

Continued from page 29.

ing entails. Developers need to keep application testability issues in mind when developing software. They need to understand, for example, how a change in a GUI control/widget implementation could affect existing automated scripts, or how logging is required for test results evaluation, etc. Project managers need to include Automated Software Testing efforts as part of the schedules and budgets. Test managers need to hire qualified Automated Software Testing personnel, and so forth. Figure 1 shows the Automated Testing Lifecycle that parallels the system engineering lifecycle.4

Automated Software Testing can be effectively applied to all software testing phases that run in parallel to the system engineering lifecycle, such as developing an automated requirements traceability (RTM) via the use of a Requirements Management System during the requirements phase; automated build verification processes that include an automated unit test during the development phase; defect tracking, test status reporting; and metrics collection during the testing phase; configuration management throughout all phases; and so forth.

Automated Software Testing Return on Investment (ROI)

If Automated Software Testing is implemented effectively, it will contribute to solving the ever increasing software testing issue. An Automated Software Testing ROI was demonstrated on an application for the Navy recently. Here is a high level description of one part of this effort:

Versions of a component used for communications onboard Navy ships and other DoD areas are delivered to Navy labs from vendors for testing and verification prior to release to respective programs and ultimately onboard war-fighters. The components each consist of nearly one million lines of code including vast complexity. Currently, it takes several months to thoroughly test multi-vendor component versions for performance, quality, and functionality. An initial Automated Software Testing implementation and ROI have shown that with Automated Software Testing substantial time savings can be achieved. (see Figure 2 ).

Figure 2 shows the initial findings: Based on the initial component testing actual results, one can project a 97% reduction in test days would occur over the course of ten years. Implementing Automated Testing to conduct testing in new and innovative ways, while shortening the testing and certification timeline, while maintaining or improving product quality, can accomplish a significant reduction in overall software development costs.

Ideally, automation in 10 years would include self-testable automated components. As for today, there are many reasons why the STL should be automated. The quality of the test

Figure 1. “Automated Testing Lifecycle parallels the System Engineering lifecycle” continues on page 31

STN 10-3: Future Directions in Software Engineering
effort is improved through automated regression testing, build verification testing, multi-platform compatibility tests, and easier ability to reproduce software problems, since automated testing takes out the human error in recreating the test steps. Test procedure development, test execution, test result analysis, documentation and status of problems should all be reduced with automated testing enabling the overall test effort and schedule to be reduced. Since Automated Software Testing applies to all phases of the STL, this would include an automated requirements traceability matrix (i.e. traceability from requirements to design/development/test cases, etc.); automated test environment setup; automated testing; automated defect tracking; etc.

Most importantly, some tests can hardly be accomplished using manual testing efforts, such as memory leak detection, stress or performance testing, high test coverage with a large amount of test data input and so on.

The challenges described related to testing complex software systems and the desire to reduce the cost and schedule associated with testing is not unique to DOD in general. Commercial businesses, large and small, are also faced with increasing large and sophisticated software projects while, at the same time, they are interested in delivering new and more capable products faster to market at the lowest possible cost. In response to these challenges, automated testing tools and methodologies have been developed and continue to emerge. In addition, the emphasis on iterative incremental development approaches where incremental software builds are utilized and incremental repetitive testing is required has further contributed to the growth in automated test tools and capabilities being utilized.

The IDC Software Research Group published report entitled, “Worldwide Distributed Automated Software Quality Tools 2005-2009 Forecast and 2004 Vendor Shares” begins by stating that the “automated software quality tools market was once again the growth leader across application life-cycle markets”. This report goes on to state, “The criticality of software to business, the increasing complexity of software applications and systems, and the relentless business pressures for quality, productivity, and faster time to market have all been positive drivers (resulting in growth in the market) and will continue to be in the foreseeable future.”

The IDC Software Research Group attributes automated software quality tools growth primarily to businesses’ desire for higher quality, increased productivity, and faster time to market.

Benefits of Automated Testing that Should Also be Considered

- Types of tests that manual testing cannot accomplish effectively, if at all – such as Concurrency, Soak, Memory Leak or Performance testing:
  
  Concurrency testing uncovers any type of concurrent user access issues, while soak and persistence testing often uncovers memory leaks when the application runs over a period of time. Automated testing tools allow for those types of tests in a stand-alone fashion, while this type of testing is very time consuming and resource intensive to conduct manually.

  Using test automation, automated scripts can be run over an extended period of time to determine whether there

Figure 2. “Example Automated Software Testing Savings over Time”

![Figure 2. “Example Automated Software Testing Savings over Time”](data:image/png;base64,)

The Case for Automated Software Testing

Continued from page 30.
is any type of performance degradation or memory leak. At the same time, timing statements can be inserted to track performance timing of each event tested. We can kick off the automated scripts on numerous PCs to simulate concurrency testing, i.e. accessing the same application resources at the same time with numerous users and monitoring for any potential issues. Automated testing tools should be used for these types of testing efforts to make them more feasible.

- **Effective Smoke (or Build Verification) Testing**
  Whenever a new software build or release is received, a test (generally referred to as “smoke test”) is run to verify that previously working functionality is still working. It sometimes can require numerous hours to complete an entire smoke test, only to determine that a faulty software build has been received, resulting in wasted testing time, because now the build has to be rejected and testing has to start all over again.

  If the smoke test is automated, the smoke test scripts could be run by the developers to verify the build quality before it is handed over to the testing team, saving valuable testing time and cost.

- **Standalone – Lights Out Testing**
  Automated testing tools can be programmed to kick off a script at a specific time.

  Consider the fact that automated testing can be run standalone and be kicked off automatically if needed, overnight, and the testers simply can analyze the results of the automated test the next day they are in the office.

- **Increased repeatability**
  Often a test is executed manually that uncovers a defect only to find out that the test cannot be repeated, i.e. the tester forgot which combinations of test steps led to the error message and is not able to reproduce the defect. Automated testing scripts take the guess work out of test repeatability.

- **Testers can focus on advanced issues**
  As tests are automated most system issues are uncovered. The automated script can be baselined and rerun for regression testing purposes, which generally yields less new defects than testing and automating new functionality. Testers can focus on newer or more advanced areas, where the most defects can be uncovered while the automated testing script verifies the regression testing area. New features are incrementally added to the automated test regression test suite.

- **Higher functional test coverage**
  Automated Testing will allow for an increase of the number of test case data combinations that manual testing could not cover. Data driven testing allowed for numerous test data combinations to be executed using one automated script. For example, in our case study during one of our prototype efforts we wanted to baseline numerous charts and routes used in an application we were testing. In order to automate this test efficiently, we only need to write one test script that calls and baselines numerous charts and routes and runs a bitmap comparison against a recorded baseline.

  Additionally, if the chart is off by just one pixel, during a manual test analysis the naked eye would probably have a difficult time detecting that pixel difference, however, the automated bitmap comparison feature in the automated testing tool will point out that difference immediately. Therefore, the accuracy of an automated test is higher in most cases.

**Automated Software Testing Pitfalls**

The above list is just a subset of the automated testing options and potential benefits. With so many benefits why are so few automated testing efforts underway or even successful? There are various reasons why automated testing efforts can fail and in many years of experience in automated testing many lessons learned have been accumulated. Here are just a few of the common mistakes programs make when implementing automated testing:

**Treating automated testing as a side activity:** It is important that automated testing is not treated as a side activity, i.e. asking a tester to automate whenever he gets free time. Testers rarely have free time and deadlines are always looming. Automated testing requires a mini-development lifecycle with test requirements, test design and test implementation and verification. Automated testing can only succeed if it is given the necessary resources required.

**Thinking anyone can automate a test:** Testing requires skills and automated testing requires software development skills. The automation effort is only successful if implemented using the appropriate expertise.

A structured approach to automated testing is necessary to help steer the test team away from some of the common test program mistakes below:

- Implementing the use of an automated test tool without a testing process in place resulting in an ad-hoc, non-repeatable, non-measurable test program
The Case for Automated Software Testing

Continued from page 32.

- Implementing a test design without following any design standards, resulting in the creation of test scripts that are not repeatable and therefore not reusable for incremental software builds
- Using the wrong tool
- Test tool implementation initiated too late in the application development life cycle, not allowing sufficient time for tool setup and test tool introduction process (i.e. learning curve)
- Test engineer involvement initiated too late in the application development life cycle resulting in poor understanding of the application and system design, which results in incomplete testing
- Not including software developers, so they can keep automated testing in mind when they make changes to the code. Developers need to understand the impact their code changes could have on an automated testing framework and can consider alternatives, as appropriate.

Automated testing enables rapid regression testing while comprehensive manual regression testing is almost prohibitive to conduct because of the time required.

Often the mistake is made to assume that a “manual” tester can pick up an automated testing tool and simply hit record and playback. However, much more is involved and a development background is required. Automated Software Testing, when done effectively, should be considered a software development effort and includes test requirements, automated test design, script development and automated script verification.

How to Automate Software Testing

Automated Testing can be accomplished using vendor provided tools, open-source tools or in-house developed tools or a combination of the above:

Vendor provided automated testing tools generally mimic the actions of the test engineer via the use of the tool’s “recording” feature. During testing, the engineer uses the keyboard and mouse to perform some type of test step or action, while the recording feature of the automated testing tool captures all keystrokes, saving the recording baselines and test results in the form of an automated test script. During subsequent test playback, scripts compare the latest test output against the previous baseline. Testing tools generally have built-in test functions, code modules, .dlls and code libraries that the test engineer can reuse. Most test tools provide for non-intrusive testing, i.e. they interact with the application-under-test without affecting the application’s behavior, as if the test tool was not involved. Vendor provided test tools use a variety of test scripting languages, i.e. Java script, VB Script, C, or vendor proprietary languages. Vendor provided tools also use various storage mechanisms, with generally no specific standard being applied across the vendor community. This type of automation can be most tedious and time-consuming with possibly the least level of Return on Investment in an environment where the application-under-test is still constantly changing.

The problem with this type of “record/playback” automation is that the script baselines contain hard coded values, i.e. if the test engineer clicks on today’s date as part of her test steps, today’s date will be recorded/baselined and trying to play back the script tomorrow (on the subsequent date) will fail. The hard coded values need to be replaced with variables, etc. The tool generated scripts generally will require much modification and coding expertise, such as understanding of the use of reusable functions/libraries, looping constructs, conditional statements, etc. Software development knowledge is required to effectively use vendor provided automated testing tools.

Open-source testing tools come in various flavors, i.e. are based on various technologies, come with different levels of capabilities and features, and can be applied to various phases of the software testing lifecycle.

Many of the open-source testing tools are becoming increasingly mature and stable enough to be safely implemented in an enterprise test environment. Implementing open-source testing tool solutions can be a viable option, especially when vendor provided tools don’t support the software engineering environment under test while the open-source tool provides the compatible features required.

In-house developed software test automation efforts are still common and are often necessary when vendor provided or open-source tools don’t meet the automated testing needs. Developing automated test scripts is a software development effort and requires a mini-software-development lifecycle.

The most successful automated testing environments develop a framework of automated tests with reusable components that is continuously maintained and new capability is added.

73% of survey respondents believe Automated Software Testing is beneficial but few automate

continues on page 34
The Case for Automated Software Testing

Continued from page 33.

While 73% of survey respondents believe Automated Testing is beneficial, 54% of the software testing survey respondents listed “lack of time” or “lack of budget” as the major reason for not automating their software testing efforts. Considering, that there doesn’t seem to be a lack of time or budget when a regression test has to be rerun manually yet again after just another showstopper has been uncovered and required a fix, no matter how long the manual regression testing cycle takes or how many testers it takes or how often it already had been run previously, isn’t it time to automate?

The second highest percentage of survey respondents listed “lack of expertise” as the reason for not automating their software testing efforts. There are various companies that provide automated testing services, plus a vast pool of automated test expertise exists that could be drawn from.

30% of survey respondents listed the regression testing phases as most time consuming.

Automated testing payoff is highest during regression testing, because at the time of regression testing the application area-under-test generally has stabilized and initial tests have been run and defects have been removed. Automated test scripts can be rerun with minimal maintenance or other involvement.

Too much time is spent on software testing. Hardware automated testing and the associated standards are prevalent in the commercial sector and have been employed successfully in the commercial arena at the various DOD organizations for many years. We need to get software testing up to par with hardware testing, which includes quick turnaround times. Implementing efficient and effective Automated Software Testing is a main step into that direction.

References

5. D. Hendrick, IDC, July 2006
8. See http://www.opensourcetesting.com for information on various open-source testing tools

About the Authors

Elfriede Dustin works at Innovative Defense Technologies (IDT) http://www.idrus.com, an Arlington based software testing consulting company, currently working on an effort to bring in automated software testing to a branch of the DOD. Elfriede is lead author of the book “Automated Software Testing,” which describes the Automated Testing Lifecycle Methodology (ATLM), a process that has been implemented at numerous companies. Elfriede is also author of various white papers, of the books “Effective Software Testing,” co-author of “The Art of Software Security Testing,” and “Quality Web Systems,” books which have been translated into many languages and are available world-wide. Dustin has been responsible for implementing automated test, or has performed as the lead consultant/manager/director guiding implementation of automated and manual software testing efforts at various commercial and Government agencies.

Bernie Gauf is President and Chief Technologist of IDT. Mr. Gauf has twenty years of experience in leading the design, development, and delivery of innovative solutions for the DoD. His experience includes the development and production of systems for passive and active sonar, electronic warfare, command and control, and computer based training and simulation for these systems. Mr. Gauf is currently leading IDT’s efforts in developing automated testing strategies and an automated testing framework suitable for DoD systems. Mr. Gauf has been invited to participate in numerous DoD panels associated with the use of COTS technology, middleware technology, and Open Architecture.

Prior to his employment at IDT, Mr. Gauf was one of the founding employees at Digital System Resources, Inc., a system integration and software company specializing in technology critical to national security and a recognized leader in providing state of the art, high quality products. DSR became one of the top 100 largest prime Department of Defense contractors for Research, Development, Test, and Evaluation through the successful transition of transformational technologies for the DoD.

Author Contact Information

Email: edustin@idrus.com
Email: bgauf@idrus.com
The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- Includes discussion forums, case studies and survey opportunities
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Current Gold Practices:

- Acquisition Process Improvement
- Architecture-First Approach
- Assess Reuse Risks and Costs
- Binary Quality Gates at the Inch-Pebble Level
- Commercial Specifications and Standards/Open Systems
- Ensure Interoperability
- Formal Inspections
- Formal Risk Management
- Goal-Question-Metric Approach
- Integrated Product and Process Development
- Model-Based Testing
- Plan for Technology Insertion
- Requirements Management
- Requirements Trade-Off/Negotiations
- Statistical Process Control
- Track Earned Value

Learn More About the DACS Gold Practice Initiative:
http://www.goldpractices.com

Become a Registered Member!
http://www.goldpractices.com/registration/register_step1.php

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

http://iac.dtic.mil/dacs
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer’s regular, hub-mounted, CD drive. You’ll use this great resource over and over again. It’s FREE, just for filling out our brief survey on the next page!

Data & Analysis Center for Software (DACS)

http://iac.dtic.mil/dacs/
Software Tech News Feedback

1. Which volume of the Software Tech News did you receive?  
STN 10:3  Future Directions in Software Engineering

2. When did you receive the newsletter? (month/year)

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)
   □ Very Satisfied  □ Satisfied  □ Neither Satisfied nor Dissatisfied  □ Dissatisfied  □ Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?
   □ Very Satisfied  □ Satisfied  □ Neither Satisfied nor Dissatisfied  □ Dissatisfied  □ Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?
   □ Very Satisfied  □ Satisfied  □ Neither Satisfied nor Dissatisfied  □ Dissatisfied  □ Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?
   □ Very Satisfied  □ Satisfied  □ Neither Satisfied nor Dissatisfied  □ Dissatisfied  □ Very Dissatisfied

7. Approximately how much of the newsletter do you read?
   □ The entire issue  □ Most of the content  □ About half the content  □ Briefly Skimmed  □ Didn’t Read

8. Would you read this newsletter in an E-mail newsletter format?
   □ Definitely  □ Probably  □ Not Sure  □ Probably Not  □ Definitely Not

9. How did you request the product or service?
   □ Phone Call  □ E-mail  □ DACS Website  □ Subscription Form  □ Other ___________________________

10. Would you recommend the Software Tech News to a colleague?
    □ Definitely  □ Probably  □ Not Sure  □ Probably Not  □ Definitely Not

11. What topics would you like to see this newsletter devoted to?
    ____________________________________________________________
    Comments (Optional)

Register for the first time            Update current subscription

Name:  Position/Title:

Organization:  Office Symbol:

Address:

City:  State:  Zip Code:

Country:  E-mail:

Telephone:  Fax:

Functional Role:

Organization Type:  □ Air Force  □ Army  □ Navy  □ Other DoD
□ Commercial  □ Non-Profit  □ Non-US  □ US Government  □ FFR&D  □ Other ___________________________

*Note: You must give us your address to receive the CD.
Letters to the Editor


Mr. Bollinger,

I finally got a chance to read the June DACS DoD Software Tech News article titled “Keeping Software Secure in a Networked World”. I am not an expert at software security, but your eight principles seem logical and useful.

However, I was a little concerned with the emphasis in principle 7 “Reward Brevity.” The text implies to me that code “bloat” is a primary source of errors that can lead to security holes. For instance the statement “Every line of code written is another opportunity for someone to make an error, and errors are far more likely to occur in new code than in old, proven code” directly implies that the fewer the number of lines of code that are written, the better. This section seems to imply that a common problem is that programmers try to write code that is readable by “anyone,” which leads to security vulnerabilities.

I agree that programmers should write code that is understandable by a skilled professional and not waste time trying to make it understandable by “anyone.” However, in my entire career, I have never experienced a single situation where code “bloat” occurred because programmers tried to write code that was understandable by someone with a relatively low level of technical expertise.

In fact, my professional experience has been exactly the opposite. Programmers seldom try to write code that is understandable by anyone other than themselves - that’s because it takes extra effort to do that. Unless the code goes through a peer review where local SMEs review the source code, programmers have no incentive to write understandable code.

The history of programming, unfortunately, has resulted in many cases in the opposite behaviour. In the early days of programming, when punch cards were the media, lines of code were costly, so programmers avoided extra lines. Comments were a burden. There was also the focus on squeezing as much “performance” out of the machine as possible because of the limited memory and speeds available. Those technical limitations are long gone, but some of their belief structures stay on with us.

The only case I’ve heard of where code bloat was a problem was when organizations measured programmer performance in terms of “lines of code...”

Continues on page 39
Letters to the Editor  Continued from page 38.

(LOC)”. That problem in the 1970's apparently caused all kinds of flaky coding practices so that a programmer could meet phony personal productivity performance standards. But my guess is that no one still uses LOC as a measure of productivity anymore, and thus it is not a common problem.

What I think your principle 7 should have focused on is to “Reward Clarity.” The code should be clear to an SME so that they can readily judge the correctness or vulnerability of the algorithms. This should include the effective use of meaningful comments or support documentation. This can include the use of effective maintenance tools like static and dynamic analyzers. It seems to me that the reality is that once the code gets in the compiler, it doesn't matter how many lines of code you write, the computer, if it has any level of code optimization at all, is going to “rewrite” the code anyway. Clarity is a much more important attribute, in my opinion, than brevity.

Not that brevity is bad, it's just that I think the focus on brevity should be on internal code structure and features and not necessarily on LOC. One of the biggest problems I’ve found with existing code is the occurrence of dead code (it can not be executed under normal circumstances) and comatose code (it might be possible to execute it, but the data or logic prevent that portion of code from ever being activated). Your Principle One talks to some degree of this issue in discussing clean floor software vs dirty floor software. But the analogy I would make is clean floor vs cluttered floor. The clutter that is in many programs is not wrong (i.e. it is not dirty per se) but it is unnecessary. This clutter comes from two primary sources: (1) dead or comatose code left in by lazy programmers or (2) unused or seldom used features which were left in the application long after they have served their purpose. These seem to me to be a much greater source of vulnerability than that caused by code bloat.

My concern with principle 7 is that it might encourage or provide an excuse to programmers who don't want to take the extra time to make their code readable by an SME. That would be the opposite of what I think you are really trying to suggest.

Thanks for the interesting article. Please note that my comments are my own opinion, and do not represent those of my employer or anyone else.

John E Moore, PhD, Systems Engineer

Author’s Reply

Mr. Moore,

I think you make a strong overall argument for “clarity” versus “bloat” being the issue, however, perhaps you’ve been in different environments. My opinions/observations are influenced by my experience with some telecom environments where bloat and diffusion of intent very much were issues. I recall that one organization had a policy of short procedures. Thus, programmers would literally take a quite clear procedure (a page or so in length) and break it up into lots of smaller pieces, which made no sense at all individually. Their perception was that they were implementing good development practices since they were following the organizational policy. In the same organization, a developer once told me “I don't see the advantage of this object oriented stuff” ... and proceeded to describe how he had converted his procedural code into a single object class with 137 methods. The methods just happened to be his original procedures.

I think that this is a pervasive issue, particularly in organizations that are not involved in defense-related coding, with its sophisticated software engineering focus.

I would like to think that LOC is no longer used much but I am troubled by anecdotal evidence indicating that is likely not the case. I suspect a lot of use of it in many settings, especially ones where the management over an effort is not particularly focused on software issues.
In This Issue

Tech Views--Challenges Dominate Our Future .... 3
Future Challenges and Rewards for Software Engineers................................. 6
Addressing the Software Engineering Challenges Over the Years and Into the Future......................................... 14
Navy C41 Open Architecture Strategy .................. 22
The Case for Automated Software Testing.......... 29

Advertisement

The Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to a paper copy, an electronic version is available at the DACS website, exposing your product, organization, or service to hundreds of thousands of additional eyes.

For rates and layout information contact: news-editor@dacs.dtic.mil